Package 'ale'

April 9, 2025

Title Interpretable Machine Learning and Statistical Inference with Accumulated Local Effects (ALE)

Version 0.5.0

Description Accumulated Local Effects (ALE) were initially developed as a model-agnostic approach for global explanations of the results of black-box machine learning algorithms. ALE has a key advantage over other approaches like partial dependency plots (PDP) and SHapley Additive exPlanations (SHAP): its values represent a clean functional decomposition of the model. As such, ALE values are not affected by the presence or absence of interactions among variables in a mode. Moreover, its computation is relatively rapid. This package reimplements the algorithms for calculating ALE data and develops highly interpretable visualizations for plotting these ALE values. It also extends the original ALE concept to add bootstrap-based confidence intervals and ALE-based statistics that can be used for statistical inference. For more details, see Okoli, Chitu. 2023. "Statistical Inference Using Machine Learning and Classical Techniques Based on Accumulated Local Effects (ALE)." arXiv. <doi:10.48550/arXiv.2310.09877>.

License MIT + file LICENSE

Language en-ca

Encoding UTF-8

RoxygenNote 7.3.2

Suggests ALEPlot, gbm, knitr, mgcv, nnet, readr, rmarkdown, testthat (>= 3.0.0), yaImpute

VignetteBuilder knitr

Imports broom, cli, dplyr, furrr, future, ggplot2, insight, methods, patchwork, progressr, purrr, rlang, S7, staccuracy, stats, stringr, tidyr, univariateML, utils

Depends R (>= 4.2.0)

URL `https://github.com/tripartio/ale`, `https://tripartio.github.io/ale/`

BugReports `https://github.com/tripartio/ale/issues`

Config/testthat/edition 3

Config/testthat/parallel true

Config/testthat/start-first ALE-numerical, ALE-binary, ALE-categorical, ModelBoot, ALEPlot-gold-standard, ALEpDist

LazyData true

NeedsCompilation no

Author Chitu Okoli [aut, cre] (<`https://orcid.org/0000-0001-5574-7572`>)

Maintainer Chitu Okoli <`Chitu.Okoli@skema.edu`>

Repository CRAN

Date/Publication 2025-04-09 20:50:02 UTC

Contents

`ALE` *ALE data and statistics that describe a trained model*

Description

An `ALE` S7 object contains ALE data and statistics. For details, see `vignette('ale-intro')` or the details and examples below.

Usage

```
ALE(
  model,
  x_cols = list(d1 = TRUE),
  data = NULL,
  y_col = NULL,
  ...,
  exclude_cols = NULL,
  parallel = "all",
  model_packages = NULL,
  output_stats = TRUE,
  output_boot_data = FALSE,
  pred_fun = function(object, newdata, type = pred_type) {
     stats::predict(object =
    object, newdata = newdata, type = type)
 },
  pred_type = "response",
  p_values = "auto",
  aler_alpha = c(0.01, 0.05),
  max_num_bins = 10,
  boot_it = 0,
  boot_alpha = 0.05,
  boot_centre = "mean",
  seed = 0,
  y_type = NULL,
  sample_size = 500,
  silent = FALSE,
  .bins = NULL
)
```

Arguments

`model`	model object. Required. Model for which ALE should be calculated. May be any kind of R object that can make predictions from data.
`x_cols`, `exclude_cols`	character, list, or formula. Columns names from `data` requested in one of the special `x_cols` formats for which ALE data is to be calculated. Defaults to 1D ALE for all columns in `data` except `y_col`. See details in the documentation for `resolve_x_cols()`.
`data`	dataframe. Dataset from which to create predictions for the ALE. It should normally be the same dataset on which `model` was trained. If not provided, `ALE()` will try to detect it automatically if it is included in the `model` object.
`y_col`	character(1). Name of the outcome target label (y) variable. If not provided, `ALE()` will try to detect it automatically from the `model` object. For non-standard models, `y_col` should be provided. For time-to-event (survival) models, see details.
`...`	not used. Inserted to require explicit naming of subsequent arguments.

`parallel`	non-negative integer(1) or character(1) in c("all", "all but one"). Number of parallel threads (workers or tasks) for parallel execution of the constructor. The default "all" uses all available physical and logical CPU cores. "all but one" uses only physical cores and reserves one core for the system. Set `parallel = 0` to disable parallel processing. See details.
`model_packages`	character. Character vector of names of packages that `model` depends on that might not be obvious with parallel processing. If you get weird error messages when parallel processing is enabled (which is the default) but they are resolved by setting `parallel = 0`, you might need to specify `model_packages`. See details.
`output_stats`	logical(1). If `TRUE` (default), return ALE statistics.
`output_boot_data`	logical(1). If `TRUE`, return the raw ALE data for each bootstrap iteration. Default is `FALSE`.
`pred_fun`, `pred_type`	function,character(1). `pred_fun` is a function that returns a vector of predicted values of type `pred_type` from `model` on `data`. See details.
`p_values`	instructions for calculating p-values. Possible values are: • `NULL`: p-values are not calculated. • An `ALEpDist` object: the object will be used to calculate p-values. • `"auto"` (default): If statistics are requested (`output_stats = TRUE`) and bootstrapping is requested (`boot_it > 0`), the constructor will try to automatically create a fast surrogate `ALEpDist` object; otherwise, no p-values are calculated. However, automatic creation of a surrogate `ALEpDist` object only works with standard R model types. If the automatic process errors, then you must explicitly create and provide an `ALEpDist()` object. Note: although faster surrogate p-values are convenient for interactive analysis, they are not acceptable for definitive conclusions or publication. See details below.
`aler_alpha`	numeric(2) from 0 to 1. Thresholds for p-values ("alpha") for confidence interval ranges for the ALER band if `p_values` are provided (that is, not `NULL`). The inner band range will be the median value of y ± `aler_alpha[2]` of the relevant ALE statistic (usually ALE range or normalized ALE range). When there is a second outer band, its range will be the median ± `aler_alpha[1]`. For example, in the ALE plots, for the default `aler_alpha = c(0.01, 0.05)`, the inner band will be the median ± ALER minimum or maximum at p = 0.05 and the outer band will be the median ± ALER minimum or maximum at p = 0.01.
`max_num_bins`	positive integer(1). Maximum number of ALE bins for numeric `x_cols` variables. The number of bins is eventually the lower of the number of unique values of a numeric variable and `max_num_bins`. Non-numeric variables such as (binary or categorical) always use all their actual values for ALE bins.
`boot_it`	non-negative integer(1). Number of bootstrap iterations for data-only bootstrapping on ALE data. This is appropriate for models that have been developed with cross-validation. For models that have not been validated, full-model bootstrapping should be used instead with a `ModelBoot()` class object. See details there. The default `boot_it = 0` turns off bootstrapping.

`boot_alpha`	numeric(1) from 0 to 1. When ALE is bootstrapped (`boot_it > 0`), `boot_alpha` specifies the thresholds for p-values ("alpha") for percentile-based confidence interval range for the bootstrapped ALE values. The bootstrap confidence intervals will be the lowest and highest `(1 - 0.05) / 2` percentiles. For example, if `boot_alpha = 0.05` (default), the confidence intervals will be from the 2.5 (low) and 97.5 (high) percentiles.
`boot_centre`	character(1) in c('mean', 'median'). When bootstrapping, the main estimate for the ALE y value is considered to be `boot_centre`. Regardless of the value specified here, both the mean and median will be available.
`seed`	integer(1). Random seed. Supply this between runs to assure that identical random ALE data is generated each time when bootstrapping. Without bootstrapping, ALE is a deterministic algorithm that should result in identical results each time regardless of the seed specified. However, with parallel processing enabled (as it is by default), only the exact computing setup will give reproducible results. For reproducible results across different computers, turn off parallelization with `parallel = 0`.
`y_type`	character(1) in c('binary', 'numeric', 'categorical', 'ordinal'). Datatype of the y (outcome) variable. Normally determined automatically; only provide if an error message for a complex non-standard model requires it.
`sample_size`	non-negative integer(1). Size of the sample of `data` to be returned with the `ALE` object. This is primarily used for rug plots in `ALEPlots()`.
`silent`	logical(1), default `FALSE`. If `TRUE`, do not display any non-essential messages during execution (such as progress bars). Regardless, any warnings and errors will always display. See details for how to customize progress bars.
`.bins`	Internal use only. List of ALE bin and n count vectors. If provided, these vectors will be used to set the intervals of the ALE x axis for each variable. By default (`NULL`), `ALE()` automatically calculates the bins. `.bins` is normally used in advanced analyses where the bins from a previous analysis are reused for subsequent analyses (for example, for full model bootstrapping with `ModelBoot()`).

Value

An object of class `ALE` with properties `effect` and `params`.

Properties

effect Stores the ALE data and, optionally, ALE statistics and bootstrap data for one or more categories.

params The parameters used to calculate the ALE data. These include most of the arguments used to construct the `ALE` object. These are either the values provided by the user or those used by default if the user did not change them but also includes several objects that are created within the constructor. These extra objects are described here, as well as those parameters that are stored differently from the form in the arguments:

```
* `max_d`: the highest dimension of ALE data present. If only 1D ALE is present, then `max_d == 1`. I
* `requested_x_cols`,`ordered_x_cols`: `requested_x_cols` is the resolved list of `x_cols` as requ
* `y_cats`: categories for categorical classification models. For non-categorical models, this is t
```

```
* `y_type`: high-level datatype of the y outcome variable.
* `y_summary`: summary statistics of y values used for the ALE calculation. These statistics are bas
* `min`, `mean`, `max`: the minimum, mean, and maximum y values, respectively. Note that the median
* `aler_lo_lo`, `aler_lo`, `aler_hi`, `aler_hi_hi`: When p-values are present, `aler_lo` and `aler
* `model`: selected elements that describe the `model` that the `ALE` object interprets.
* `data`: selected elements that describe the `data` used to produce the `ALE` object. To avoid the
```

Custom predict function

The calculation of ALE requires modifying several values of the original `data`. Thus, `ALE()` needs direct access to the `predict` function for the `model`. By default, `ALE()` uses a generic default `predict` function of the form `predict(object, newdata, type)` with the default prediction type of `'response'`. If, however, the desired prediction values are not generated with that format, the user must specify what they want. Very often, the only modification needed is to change the prediction type to some other value by setting the `pred_type` argument (e.g., to `'prob'` to generated classification probabilities). But if the desired predictions need a different function signature, then the user must create a custom prediction function and pass it to `pred_fun`. The requirements for this custom function are:

- It must take three required arguments and nothing else:
  - `object`: a model
  - `newdata`: a dataframe or compatible table type such as a tibble or data.table
  - `type`: a string; it should usually be specified as `type = pred_type` These argument names are according to the R convention for the generic `stats::predict()` function.
- It must return a vector or matrix of numeric values as the prediction.

You can see an example below of a custom prediction function.

ALE statistics and p-values

For details about the ALE-based statistics (ALED, ALER, NALED, and NALER), see `vignette('ale-statistics')`. For general details about the calculation of p-values, see `ALEpDist()`. Here, we clarify the automatic calculation of p-values with the `ALE()` constructor.

As explained in the documentation above for the `p_values` argument, the default `p_values = "auto"` will try to automatically create a fast surrogate `ALEpDist` object. However, this is on the condition that statistics are requested (default, `output_stats = TRUE`) and bootstrapping is also requested (not default, if `boot_it` is any value greater than 0). Requesting statistics is necessary otherwise p-values are not needed. However, the requirement for requiring bootstrapping is a pragmatic design choice. The challenge is that creating an `ALEpDist` object can be slow. (Even the fast surrogate option rarely takes less than 10 seconds, even with parallelization.) Thus, to optimize speed, p-values will not be calculated unless requested. However, if the user requests bootstrapping (which is slower than not requesting it), it can be assumed that they are willing to sacrifice some speed for the sake of greater precision in their ALE analysis; thus, extra time is taken to at least create a relatively faster surrogate `ALEpDist` object.

Parallel processing

Parallel processing using the `{furrr}` framework is enabled by default. The number of parallel threads (workers or cores) is specified with the `parallel` argument. By default (`parallel`

`= "all"`), it will use all the available physical and logical CPU cores. However, if the procedure is very slow (with a large dataset and slow prediction algorithm), you might want to set `parallel = "all but one"`), which will only use faster physical cores and reserve one physical core so that your computer does not slow down as you continue working on other tasks while the procedure runs. To disable parallel processing, set `parallel = 0`.

The `{ale}` package should be able to automatically recognize and load most packages that are needed, but with parallel processing enabled (which is the default), some packages might not be properly loaded. This problem might be indicated if you get a strange error message that mentions something somewhere about "progress interrupted" or "future", especially if you see such errors after the progress bars begin displaying (assuming you did not disable progress bars with `silent = TRUE`). In that case, first try disabling parallel processing with `parallel = 0`. If that resolves the problem, then to get faster parallel processing to work, try adding all the package names needed for the `model` to the `model_packages` argument, e.g., `model_packages = c('tidymodels', 'mgcv')`.

Time-to-event (survival) models

For time-to-event (survival) models, set the following arguments:

- `y_col` must be the set to the name of the binary event column.
- Include the time column in the `exclude_cols` argument so that its ALE will not be calculated, e.g., `exclude_cols = 'time'`. This is not essential but if it is not excluded, it will always result in an exactly zero ALE effect because time is an outcome, not a predictor, of the time-to-event model's outcome, so calculating it is a waste of time.
- `pred_type` must be specified according to the desired `type` argument for the `predict()` method of the time-to-event algorithm (e.g., "risk", "survival", "time", etc.).
- `pred_fun` might work fine without modification as long as the settings above are configured. However, for non-standard time-to-event models, a custom `pred_fun` as specified above might be needed.

Progress bars

Progress bars are implemented with the `{progressr}` package. For details on customizing the progress bars, see the introduction to the `{progressr}` package. To disable progress bars when calling a function in the `ale` package, set `silent = TRUE`.

References

Okoli, Chitu. 2023. “Statistical Inference Using Machine Learning and Classical Techniques Based on Accumulated Local Effects (ALE).” arXiv. `doi:10.48550/arXiv.2310.09877`.

Examples

```
# Sample 1000 rows from the ggplot2::diamonds dataset (for a simple example)
set.seed(0)
diamonds_sample <- ggplot2::diamonds[sample(nrow(ggplot2::diamonds), 1000), ]

# Create a GAM model with flexible curves to predict diamond price
# Smooth all numeric variables and include all other variables
gam_diamonds <- mgcv::gam(
```

```
  price ~ s(carat) + s(depth) + s(table) + s(x) + s(y) + s(z) +
    cut + color + clarity +
    ti(carat, by = clarity),  # a 2D interaction
  data = diamonds_sample
)
summary(gam_diamonds)




# Simple ALE without bootstrapping: by default, all 1D ALE effects
ale_gam_diamonds <- ALE(gam_diamonds)

# Simple printing of all plots
plot(ale_gam_diamonds)

# Bootstrapped ALE
# This can be slow, since bootstrapping runs the algorithm boot_it times

# Create ALE with 100 bootstrap samples
ale_gam_diamonds_boot <- ALE(
  gam_diamonds,
  # request all 1D ALE effects and only the carat:clarity 2D effect
  list(d1 = TRUE, d2 = 'carat:clarity'),
  boot_it = 100
)

#' More advanced plot manipulation
ale_plots <- plot(ale_gam_diamonds_boot) # Create an ALEPlots object

# Print the plots: First page prints 1D ALE; second page prints 2D ALE
ale_plots  # or print(ale_plots) to be explicit

# Extract specific plots (as lists of ggplot objects)
get(ale_plots, 'carat')  # extract a specific 1D plot
get(ale_plots, 'carat:clarity')  # extract a specific 2D plot
get(ale_plots, type = 'effect')  # ALE effects plot
# See help(get.ALEPlots) for more options, such as for categorical plots




# If the predict function you want is non-standard, you may define a
# custom predict function. It must return a single numeric vector.
custom_predict <- function(object, newdata, type = pred_type) {
  predict(object, newdata, type = type, se.fit = TRUE)$fit
}

ale_gam_diamonds_custom <- ALE(
  gam_diamonds,
  pred_fun = custom_predict, pred_type = 'link'
)

# Plot the ALE data
```

```
plot(ale_gam_diamonds_custom)


# How to retrieve specific types of ALE data from an ALE object.
ale_diamonds_with_boot_data <- ALE(
  gam_diamonds,
  # For detailed options for x_cols, see examples at resolve_x_cols()
  x_cols = ~ carat + cut + clarity + carat:clarity + color:depth,
  output_boot_data = TRUE,
  boot_it = 10  # just for demonstration
)

# See ?get.ALE for details on the various kinds of data that may be retrieved.
get(ale_diamonds_with_boot_data, ~ carat + color:depth)  # default ALE data
get(ale_diamonds_with_boot_data, what = 'boot_data')  # raw bootstrap data
get(ale_diamonds_with_boot_data, stats = 'estimate')  # summary statistics
get(ale_diamonds_with_boot_data, stats = c('aled', 'naled'))
get(ale_diamonds_with_boot_data, stats = 'all')
get(ale_diamonds_with_boot_data, stats = 'conf_regions')
get(ale_diamonds_with_boot_data, stats = 'conf_sig')
```

`ALEpDist` *Random variable distributions of ALE statistics for generating p-values*

Description

ALE statistics are accompanied with two indicators of the confidence of their values. First, bootstrapping creates confidence intervals for ALE effects and ALE statistics to give a range of the possible ALE values. Second, we calculate p-values, an indicator of the probability that a given ALE statistic is random. An `ALEpDist` S7 object contains the necessary distribution data for generating such p-values.

Usage

```
ALEpDist(
  model,
  data = NULL,
  ...,
  y_col = NULL,
  rand_it = NULL,
  surrogate = FALSE,
  parallel = "all",
  model_packages = NULL,
  random_model_call_string = NULL,
  random_model_call_string_vars = character(),
```

```
  positive = TRUE,
  pred_fun = function(object, newdata, type = pred_type) {
     stats::predict(object =
    object, newdata = newdata, type = type)
 },
  pred_type = "response",
  output_residuals = FALSE,
  seed = 0,
  silent = FALSE,
  .skip_validation = FALSE
)
```

Arguments

`model`	See documentation for `ALE()`
`data`	See documentation for `ALE()`
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`y_col`	See documentation for `ALE()`
`rand_it`	non-negative integer(1). Number of times that the model should be retrained with a new random variable. The default of `NULL` will generate 1000 iterations, which should give reasonably stable p-values; these are considered "exact" p-values. It can be reduced for approximate ("approx") p-values as low as 100 for faster test runs but then the p-values are not as stable. `rand_it` below 100 is not allowed as such p-values are inaccurate.
`surrogate`	logical(1). Create p-value distributions based on a surrogate linear model (`TRUE`) instead of on the original `model` (default `FALSE`). Note that while faster surrogate p-values are convenient for interactive analysis, they are not acceptable for definitive conclusions or publication. See details.
`parallel`	See documentation for `ALE()`. Note that for exact p-values, by default 1000 random variables are trained. So, even with parallel processing, the procedure is very slow.
`model_packages`	See documentation for `ALE()`
`random_model_call_string`	character(1). If `NULL`, the `ALEpDist()` constructor tries to automatically detect and construct the call for p-values. If it cannot, the constructor will fail. In that case, a character string of the full call for the model must be provided that includes the random variable. See details.
`random_model_call_string_vars`	See documentation for `model_call_string_vars` in `ModelBoot()`; their operation is very similar.
`positive`	See documentation for `ModelBoot()`
`pred_fun, pred_type`	See documentation for `ALE()`
`output_residuals`	logical(1). If `TRUE`, returns the residuals in addition to the raw data of the generated random statistics (which are always returned). The default `FALSE` does not return the residuals.

`seed`	See documentation for `ALE()`
`silent`	See documentation for `ALE()`
`.skip_validation`	Internal use only. logical(1). Skip non-mutating data validation checks. Changing the default `FALSE` risks crashing with incomprehensible error messages.

Value

An object of class `ALEpDist` with properties `rand_stats`, `residual_distribution`, `residuals`, and `params`.

Properties

rand_stats A named list of tibbles. There is normally one element whose name is the same as `y_col` except if `y_col` is a categorical variable; in that case, the elements are named for each category of `y_col`. Each element is a tibble whose rows are each of the `rand_it_ok` iterations of the random variable analysis and whose columns are the ALE statistics obtained for each random variable.

residual_distribution A `univariateML` object with the closest estimated distribution for the `residuals` as determined by `univariateML::model_select()`. This is the distribution used to generate all the random variables.

residuals If `output_residuals == TRUE`, returns a matrix of the actual `y_col` values from `data` minus the predicted values from the `model` (without random variables) on the `data`. The rows correspond to each row of `data`. The columns correspond to the named elements (`y_col` or categories) described above for `rand_stats`. `NULL` if `output_residuals == FALSE` (default).

params Parameters used to generate p-value distributions. Most of these repeat selected arguments passed to `ALEpDist()`. These are either values provided by the user or used by default if the user did not change them but the following additional or modified objects are notable:

```
* `model`: selected elements that describe the `model` used to generate the random distributions.
* `rand_it`: the number of random iterations requested by the user either explicitly (by specifying
* `rand_it_ok`: A whole number with the number of `rand_it` iterations that successfully generated
* `exactness`: A string. For regular p-values generated from the original model, `'exact'` if `rand
```

Exact p-values for ALE statistics

Because ALE is non-parametric (that is, it does not assume any particular distribution of data), the `{ale}` package takes a literal frequentist approach to the calculation of empirical (Monte Carlo) p-values. That is, it literally retrains the model 1000 times, each time modifying it by adding a distinct random variable to the model. (The number of iterations is customizable with the `rand_it` argument.) The ALEs and ALE statistics are calculated for each random variable. The percentiles of the distribution of these random-variable ALEs are then used to determine p-values for non-random variables. Thus, p-values are interpreted as the frequency of random variable ALE statistics that exceed the value of ALE statistic of the actual variable in question. The specific steps are as follows:

- The residuals of the original model trained on the training data are calculated (residuals are the actual y target value minus the predicted values).

- The closest distribution of the residuals is detected with `univariateML::model_select()`.
- 1000 new models are trained by generating a random variable each time with `univariateML::rml()` and then training a new model with that random variable added.
- The ALEs and ALE statistics are calculated for each random variable.
- For each ALE statistic, the empirical cumulative distribution function (`stats::ecdf()`) is used to create a function to determine p-values according to the distribution of the random variables' ALE statistics.

Because the `ale` package is model-agnostic (that is, it works with any kind of R model), the `ALEpDist()` constructor cannot always automatically manipulate the model object to create the p-values. It can only do so for models that follow the standard R statistical modelling conventions, which includes almost all base R algorithms (like `stats::lm()` and `stats::glm()`) and many widely used statistics packages (like `mgcv` and `survival`), but which excludes most machine learning algorithms (like `tidymodels` and `caret`). For non-standard algorithms, the user needs to do a little work to help the `ALEpDist()` constructor correctly manipulate its model object:

- The full model call must be passed as a character string in the argument `random_model_call_string`, with two slight modifications as follows.
- In the formula that specifies the model, you must add a variable named 'random_variable'. This corresponds to the random variables that the constructor will use to estimate p-values.
- The dataset on which the model is trained must be named 'rand_data'. This corresponds to the modified datasets that will be used to train the random variables.

See the example below for how this is implemented.

If the model generation is unstable (because of a small dataset size or a finicky model algorithm), then one or more iterations might fail, possibly dropping the number of successful iterations to below 1000. Then the p-values are only considered approximate; they are no longer exact. If that is the case, then request rand_it at a sufficiently high number such that even if some iterations fail, at least 1000 will succeed. For example, for an `ALEpDist` object named `p_dist`, if `p_dist@params$rand_it_ok` is 950, you could rerun `ALEpDist()` with `rand_it = 1100` or higher to allow for up to 100 possible failures.

Faster approximate and surrogate p-values

The procedure we have just described requires at least 1000 random iterations for p-values to be considered **"exact"**. Unfortunately, this procedure is rather slow–it takes at least 1000 times as long as the time it takes to train the model once.

With fewer iterations (at least 100), p-values can only be considered **approximate ("approx")**. Fewer than 100 such p-values are invalid. There might be fewer iterations either because the user requests them with the `rand_it` argument or because some iterations fail for whatever reason. As long as at least 1000 iterations succeed, p-values will be considered exact.

Because the procedure can be very slow, a faster version of the algorithm generates **"surrogate"** p-values by substituting the original `model` with a linear model that predicts the same `y_col` outcome from all the other columns in `data`. By default, these surrogate p-values use only 100 iterations and if the dataset is large, the surrogate model samples 1000 rows. Thus, the surrogate p-values can be generated much faster than for slower model algorithms on larger datasets. Although they are suitable for model development and analysis because they are faster to generate, they are less reliable than approximate p-values based on the original model. In any case, **definitive conclusions (e.g., for publication) always require exact p-values with at least 1000 iterations on the original model**. Note that surrogate p-values are always marked as "surrogate"; even if they are generated based on over 1000 iterations, they can never be considered exact because they are not based on the original `model`.

References

Okoli, Chitu. 2023. "Statistical Inference Using Machine Learning and Classical Techniques Based on Accumulated Local Effects (ALE)." arXiv. doi:10.48550/arXiv.2310.09877.

Examples

```
# Sample 1000 rows from the ggplot2::diamonds dataset (for a simple example)
set.seed(0)
diamonds_sample <- ggplot2::diamonds[sample(nrow(ggplot2::diamonds), 1000), ]

# Create a GAM with flexible curves to predict diamond price
# Smooth all numeric variables and include all other variables
gam_diamonds <- mgcv::gam(
  price ~ s(carat) + s(depth) + s(table) + s(x) + s(y) + s(z) +
    cut + color + clarity +
    ti(carat, by = clarity),  # a 2D interaction
  data = diamonds_sample
)
summary(gam_diamonds)

# Create p_value distribution
pd_diamonds <- ALEpDist(
  gam_diamonds,
  diamonds_sample,
  # only 100 iterations for a quick demo; but usually should remain at 1000
  rand_it = 100
)

# Examine the structure of the returned object
print(pd_diamonds)
# In RStudio: View(pd_diamonds)

# Calculate ALEs with p-values
ale_gam_diamonds <- ALE(
  gam_diamonds,
  p_values = pd_diamonds
)

# Plot the ALE data. The horizontal bands in the plots use the p-values.
plot(ale_gam_diamonds)


# For non-standard models that give errors with the default settings,
# you can use 'random_model_call_string' to specify a model for the estimation
# of p-values from random variables as in this example.
```

```
# See details above for an explanation.
pd_diamonds <- ALEpDist(
  gam_diamonds,
  diamonds_sample,
  random_model_call_string = 'mgcv::gam(
    price ~ s(carat) + s(depth) + s(table) + s(x) + s(y) + s(z) +
        cut + color + clarity + random_variable,
    data = rand_data
  )',
  # only 100 iterations for a quick demo; but usually should remain at 1000
  rand_it = 100
)

# Examine the structure of the returned object
print(pd_diamonds)
# In RStudio: View(pd_diamonds)
```

`ALEPlots` *ALE plots with print and plot methods*

Description

An `ALEPlots` S7 object contains the ALE plots from `ALE` or `ModelBoot` objects stored as `ggplot` objects. The `ALEPlots` constructor creates all possible plots from the `ALE` or `ModelBoot` passed to it—not only individual 1D and 2D ALE plots, but also special plots like the ALE effects plot. So, an `ALEPlots` object is a collection of plots, almost never a single plot. To retrieve specific plots, use the `get.ALEPlots()` method. See the examples with the `ALE()` and `ModelBoot()` objects for how to manipulate `ALEPlots` objects.

Usage

```
ALEPlots(
  obj,
  ...,
  ale_centre = "median",
  y_1d_refs = c("25%", "75%"),
  rug_sample_size = obj@params$sample_size,
  min_rug_per_interval = 1,
  y_nonsig_band = 0.05,
  seed = 0,
  silent = FALSE
)
```

Arguments

`obj`	`ALE` or `ModelBoot` object. The object containing ALE data to be plotted.
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`ale_centre`	character(1) in c('median', 'mean', 'zero'). The ALE y values in the plots will be centred relative to this value. 'median' is the default. 'zero' will maintain the actual ALE values, which are centred on zero.
`y_1d_refs`	character or numeric vector. For 1D ALE plots, the y outcome values for which a reference line should be drawn. If a character vector, `y_1d_refs` values are names from `obj@params$y_summary` (usually quantile names). If a numeric vector, `y_1d_refs` values must be values within the range of y, that is, between `obj@params$y_summary$min` and `obj@params$y_summary$max` inclusive.
`rug_sample_size`, `min_rug_per_interval`	non-negative integer(1). Rug plots are down-sampled to `rug_sample_size` rows, otherwise they can be very slow for large datasets. By default, their size is the value of `obj@params$sample_size`. They maintain representativeness of the data by guaranteeing that each of the ALE bins will retain at least `min_rug_per_interval` elements; usually set to just 1 (default) or 2. To prevent this down-sampling, set `rug_sample_size` to `Inf` (but then the `ALEPlots` object would store the entire dataset, so could become very large).
`y_nonsig_band`	numeric(1) from 0 to 1. If there are no p-values, some plots (notably the 1D effects plot) will shade grey the inner `y_nonsig_band` quantile below and above the `ale_centre` average (the median, by default) to indicate nonsignificant effects.
`seed`	See documentation for `ALE()`
`silent`	See documentation for `ALE()`

Value

An object of class `ALEPlots` with properties `plots` and `params`.

Properties

plots Stores the ALE plots. Use `get.ALEPlots()` to access them.

params The parameters used to calculate the ALE plots. These include most of the arguments used to construct the `ALEPlots` object. These are either the values provided by the user or used by default if the user did not change them but also includes several objects that are created within the constructor. These extra objects are described here, as well as those parameters that are stored differently from the form in the arguments:

```
* `y_col`, `y_cats`: See documentation for [ALE()]
* `max_d`: See documentation for [ALE()]
* `requested_x_cols`: See documentation for [ALE()]. Note, however, that `ALEPlots` does not store
```

Examples

```
# See examples with ALE and ModelBoot objects
```

`census` *Census Income*

Description

Census data that indicates, among other details, if the respondent's income exceeds $50,000 per year. Also known as "Adult" dataset.

Usage

```
census
```

Format

A tibble with 32,561 rows and 15 columns:

higher_income TRUE if income > $50,000

age continuous

workclass Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked

fnlwgt continuous. "A proxy for the demographic background of the people: 'People with similar demographic characteristics should have similar weights'" For more details, see https://www.openml.org/search?type=d

education Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool

education_num continuous

marital_status Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse

occupation Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces

relationship Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried

race White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black

sex Female, Male

capital_gain continuous

capital_loss continuous

hours_per_week continuous

native_country United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinidad&Tobago, Peru, Hong, Holland-Netherlands

This dataset is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

Source

Becker,Barry and Kohavi,Ronny. (1996). Adult. UCI Machine Learning Repository. https://doi.org/10.24432/C5XW20.

`get` *S7 generic get method for objects in the ale package*

Description

Retrieve specific data elements from an object based on their X column names.

If `obj` is not an object from the `ale` package, then this generic passes on all arguments to the `base::get()` function.

Usage

```
get(obj, ...)
```

Arguments

`obj`	object.
`...`	For ale package objects, instructions for which predictor (x) columns should be retrieved. For everything else, arguments to pass to `base::get()`.

`get.ALE` *get method for ALE objects*

Description

Retrieve specific elements from an `ALE` object.

Arguments

`obj`	ALE object from which to retrieve elements.
`x_cols, exclude_cols`	character, list, or formula. Columns names and interaction terms from `obj` requested in one of the special `x_cols` formats. The default value of `NULL` for `x_cols` retrieves all available data of the output requested in `what`. See details in the documentation for `resolve_x_cols()`.
`what`	character(1). What kind of output is requested. Must be either "ale" (default) or "boot_data". To retrieve ALE statistics, see the `stats` argument.
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`stats`	character(1). Retrieve ALE statistics. If `stats` is specified, then `what` must be left at the default ("ale"). Otherwise, `get()` errors if `stats` is specified and `what` has some other value. See the return value details below for valid values for `stats`.

`cats`	character. Optional category names to retrieve if the ALE is for a categorical y outcome model.
`ale_centre`	Same as in documentation for `ALEPlots()`
`simplify`	logical(1). If `TRUE` (default), the results will be simplified to the simplest list structure possible to give the requested results. If `FALSE`, a complex but consistent list structure will be returned; this might be preferred for programmatic and non-interactive use.
`silent`	See documentation for `resolve_x_cols()`

Value

Regardless of the requested data, all `get.ALE()` have a common structure:

- If more than one category of the y outcome is returned, then the top level is a list named by each category. If, however, the y outcome is not categorical or only one category of multiple possibilities is specified using the `cats` argument, then the top level never has categories, regardless of the value of `simplify`.
- The next level (or top level if there are zero or one category) is a list with one or two levels:
  - `d1`: 1D ALE elements.
  - `d2`: 2D ALE elements. However, if elements of only one dimension (either 1D or 2D) are requested and `simplify = TRUE` (default), the empty list is eliminated and the level is skipped to provide only the elements present. For example, if only 1D ALE data is requested, then there will be no `d1` sublist but only a list of the ALE data as described for the next level. If `simplify = FALSE`, both `d1` and `d2` sublists will always be returned; the empty sublist will be `NULL`.

While all results follow the general structure just described, the specific type of data returned depends on the values of the `what` and `stats` arguments:

`what = 'ale'` (default) and `stats = NULL` (default) A list whose elements, named by each requested x variable, are each a tibble. The rows each represent one ALE bin. The tibble has the following columns: * `var.bin` or `var.ceil` where `var` is the name of a variable (column): For non-numeric x, `var.bin` is the value of each of the ALE categories. For numeric x, `var.ceil` is the value of the upper bound (ceiling) of each ALE bin. The first "bin" of numeric variables represents the minimum value. For 2D ALE with an `var1` by `var2` interaction, both `var1.bin` and `var2.bin` columns are returned (or `var1.ceil` or `var2.ceil`for numeric `var1` or `var2`). * `.n`: the number of rows of data in each bin represented by `var.bin` or `var.ceil`. For numeric x, the first bin contains all data elements that have exactly the minimum value of x. This is often 1, but might be more than 1 if more than one data element has exactly the minimum value. * `.y`: the ALE function value calculated for that bin. For bootstrapped ALE, this is the same as `.y_mean` by default or `.y_median` if `boot_centre = 'median'`. Regardless, both `.y_mean` and `.y_median` are returned as columns here. * `.y_lo`, `.y_hi`: the lower and upper confidence intervals, respectively, for the bootstrapped `.y` value based on the `boot_alpha` argument in the `ALE()` constructor.

`what = 'boot_data'` and `stats = NULL` (default) A list whose elements, named by each requested x variable, are each a tibble. These are the data from which `.y_mean`, `.y_median`, `.y_lo`, and `.y_hi` are summarized when `what = 'ale'`. The rows each represent one ALE bin for a specified bootstrap iteration. The tibble has the following columns: * `.it`: The bootstrap iteration.

Iteration 0 represents the ALE calculations on the full dataset; the remaining values of `.it` are from 1 to `boot_it` (number of bootstrap iterations specified in the `ALE()` constructor. * `var` where `var` is the name of a variable (column): For non-numeric x, `var` is the value of each of the ALE categories. For numeric x, `var` is the value of the upper bound (ceiling) of each ALE bin. They are otherwise similar to their meanings described for `what = 'ale'` above. * `.n` and `.y`: Same as for `what = 'ale'`.

`what = ’ale’` **(default) and** `stats = ’estimate’` A list with elements `d1` and `d2` with the value of each ALE statistic. Each row represents one variable or interaction. The tibble has the following columns: * `term`: The variables or columns for the 1D or 2D ALE statistic. * `aled`, `aler_min`, `aler_max`, `naled`, `naler_min`, `naler_max`: the respective ALE statistic for the variable or interaction.

`what = ’ale’` **(default) and** `stats` **is one or more values in** `c(’aled’, ’aler_min’, ’aler_max’, ’naled’, ’naler_mi`
A list with elements `d1` and `d2` with the distribution value of the single requested ALE statistic. Each element `d1` and `d2` is a tibble. Each row represents one statistic for one variable or interaction. The tibble has the following columns: * `term`: Same as for `stats = 'estimate'`. * `statistic`: The requested ALE statistic(s). * `estimate`, `mean`, `median`: The average of the bootstrapped value of the requested statistic. `estimate` is equal to either `mean` or `median` depending on the `boot_centre` argument in the `ALE()` constructor. If ALE is not bootstrapped, then `estimate`, `mean`, and `median` are equal. * `conf.low`, `conf.high`: the lower and upper confidence intervals, respectively, for the bootstrapped statistic based on the `boot_alpha` argument in the `ALE()` constructor. If ALE is not bootstrapped, then `estimate`, `conf.low`, and `conf.high` are equal.

`what = ’ale’` **(default) and** `stats = ’all’` A list with elements `d1` and `d2` with the distribution values of all available ALE statistics for the requested variables and interactions. Whereas the `stats = 'aled'` (for example) format returns data for a single statistic, `stats = 'all'` returns all statistics for the requested variables. Thus, the data structure and columns are identical as for single statistics above, except that all available ALE statistics are returned.

`what = ’ale’` **(default) and** `stats = ’conf_regions’` A list with elements `d1` and `d2` with the confidence regions for the requested variables and interactions. Each element is a list with the requested `d1` and `d2` sub-elements as described in the general structure above. Each data element is a tibble with confidence regions for a single variable or interaction. For an explanation of the columns, see `vignette('ale-statistics')`.

`what = ’ale’` **(default) and** `stats = ’conf_sig’` Identical structure as `stats = 'conf_regions'` except that the elements are filtered for the terms (variables or interactions) that have statistically significant confidence regions exceeding the threshold of the inner ALER band, specifically, at least `obj@params$aler_alpha[2]` of the rows of data. See `vignette("ale-statistics")` for details.

Examples

```
# See examples at ALE() for a demonstration of how to use the get() method.
```

`get.ALEPlots` *get method for ALEPlots objects*

Description

Retrieve specific plots from a `ALEPlots` object. Unlike `subset.ALEPlots()` which returns an `ALEPlots` object with the subsetted `x_cols` variables and interactions, this `get.ALEPlots()` method returns a list of `ggplot2::ggplot` objects as specified in the return value description. To retain special `ALEPlots` behaviour like plotting, printing, and summarizing multiple plots, use `subset.ALEPlots()` instead.

See `get.ALE()` for explanation of parameters not described here.

Arguments

`obj`	ALEPlots object from which to retrieve ALE elements.
`type`	character(1). What type of ALEPlots to retrieve: `'ale'` for standard ALE plots or `'effect'` for ALE effects plots. See `cats` argument for options for categorical plots.
`cats`	character. The categories (one or more) of a categorical outcome variable to retrieve. To retrieve all categories as individual category plots, leave `cats` at the default `NULL`. For categorical plots that combine all categories, specify `cats = ".all"`. (Don't forget the "." in ".all", which avoids naming conflicts with legitimate categories that might be named "all".) For such all-category plots, `type` must be set to "overlay" or "facet" for the specific desired type of categorical plot.

Value

A list of `ggplot` objects as described in the documentation for the return value of `get.ALE()`. This is different from `subset.ALEPlots()`, which returns an `ALEPlots` object with the subsetted `x_cols` variables and interactions.

`get.ModelBoot` *get method for ModelBoot objects*

Description

Retrieve specific ALE elements from a `ModelBoot` object. This method is similar to `get.ALE()` except that the user may specify what `type` of ALE data to retrieve (see the argument definition for details).

See `get.ALE()` for explanation of parameters not described here.

Arguments

`obj`	ModelBoot object from which to retrieve ALE elements.
`type`	character(1). The type of ModelBoot ALE elements to retrieve: `'single'` for the ALE calculated on the full data set or `'boot'` for the bootstrapped ALE data (based on full-model bootstrapping). The default `'auto'` will retrieve `'boot'` if it is available and `'single'` otherwise.

Value

See `get.ALE()`

`ModelBoot` *Statistics and ALE data for a bootstrapped model*

Description

A `ModelBoot` S7 object contains full-model bootstrapped statistics and ALE data for a trained model. Full-model bootstrapping (as distinct from data-only bootstrapping) retrains a model for each bootstrap iteration. Thus, it can be rather slow, though it is much more reliable. However, for obtaining bootstrapped ALE data, plots, and statistics, full-model bootstrapping as provided by `ModelBoot` is only necessary for models that have not been developed by cross-validation. For cross-validated models, it is sufficient (and much faster) to create a regular `[ALE()]` object with bootstrapping by setting the `boot_it` argument in its constructor. In fact, full-model bootstrapping with `ModelBoot` is often infeasible for slow machine-learning models trained on large datasets, which should rather be cross-validated to assure their reliability. However, for models that have not been cross-validated, full-model bootstrapping with `ModelBoot` is necessary for reliable results. Further details follow below; see also `vignette('ale-statistics')`.

Usage

```
ModelBoot(
  model,
  data = NULL,
  ...,
  model_call_string = NULL,
  model_call_string_vars = character(),
  parallel = "all",
  model_packages = NULL,
  y_col = NULL,
  positive = TRUE,
  pred_fun = function(object, newdata, type = pred_type) {
     stats::predict(object =
    object, newdata = newdata, type = type)
 },
  pred_type = "response",
  boot_it = 100,
```

```
  boot_alpha = 0.05,
  boot_centre = "mean",
  seed = 0,
  output_model_stats = TRUE,
  output_model_coefs = TRUE,
  output_ale = TRUE,
  output_boot_data = FALSE,
  ale_options = list(),
  ale_p = "auto",
  tidy_options = list(),
  glance_options = list(),
  silent = FALSE
)
```

Arguments

`model`	Required. See documentation for `ALE()`
`data`	dataframe. Dataset to be bootstrapped. This must be the same data on which the `model` was trained. If not provided, `ModelBoot()` will try to detect it automatically. For non-standard models, `data` should be provided.
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`model_call_string`	character(1). If `NULL` (default), the `ModelBoot` tries to automatically detect and construct the call for bootstrapped datasets. If it cannot, the function will fail early. In that case, a character string of the full call for the model must be provided that includes `boot_data` as the data argument for the call. See examples.
`model_call_string_vars`	character. Names of variables included in `model_call_string` that are not columns in `data`. If any such variables exist, they must be specified here or else parallel processing may produce an error. If parallelization is disabled with `parallel = 0`, then this is not a concern. See documentation for the `model_packages` argument in `ALE()`.
`parallel`, `model_packages`	See documentation for `ALE()`
`y_col`, `pred_fun`, `pred_type`	See documentation for `ALE()`. Used to calculate bootstrapped performance measures. If left at their default values, then the relevant performance measures are calculated only if these arguments can be automatically detected. Otherwise, they should be specified.
`positive`	any single atomic value. If the model represented by `model` or `model_call_string` is a binary classification model, `positive` specifies the 'positive' value of `y_col` (the target outcome), that is, the value of interest that is considered `TRUE`; any other value of `y_col` is considered `FALSE`. This argument is ignored if the model is not a binary classification model. For example, if 2 means `TRUE` and 1 means `FALSE`, then set `positive = 2`.
`boot_it`	non-negative integer(1). Number of bootstrap iterations for full-model bootstrapping. For bootstrapping of ALE values, see details to verify if `ALE()` with

	bootstrapping is not more appropriate than `ModelBoot()`. If `boot_it = 0`, then the model is run as normal once on the full `data` with no bootstrapping.
`boot_alpha`	numeric(1) from 0 to 1. Alpha for percentile-based confidence interval range for the bootstrap intervals; the bootstrap confidence intervals will be the lowest and highest `(1 - 0.05) / 2` percentiles. For example, if `boot_alpha = 0.05` (default), the intervals will be from the 2.5 and 97.5 percentiles.
`boot_centre`	character(1) in c('mean', 'median'). When bootstrapping, the main estimate for the ALE y value is considered to be `boot_centre`. Regardless of the value specified here, both the mean and median will be available.
`seed`	integer. Random seed. Supply this between runs to assure identical bootstrap samples are generated each time on the same data. See documentation for `ALE()` for further details.
`output_model_stats`	logical(1). If `TRUE` (default), return overall model statistics using `broom::glance()` (if available for `model`) and bootstrap-validated statistics if `boot_it > 0`.
`output_model_coefs`	logical(1). If `TRUE` (default), return model coefficients using `broom::tidy()` (if available for `model`).
`output_ale`	logical(1). If `TRUE` (default), return ALE data and statistics.
`output_boot_data`	logical(1). If `TRUE`, return the full raw data for each bootstrap iteration, specifically, the bootstrapped models and the model row indices. Default `FALSE` does not return this large, detailed data.
`ale_options`, `tidy_options`, `glance_options`	list of named arguments. Arguments to pass to the `ALE()` constructor when `ale = TRUE`, `broom::tidy()` when `model_coefs = TRUE`, or `broom::glance()` when `model_stats = TRUE`, respectively, beyond (or overriding) their defaults. Note: to obtain p-values for ALE statistics, see the `ale_p` argument.
`ale_p`	Same as the `p_values` argument for the `ALE()` constructor; see documentation there. This argument overrides the `p_values` element of the `ale_options` argument.
`silent`	See documentation for `ALE()`

Value

An object of class `ALE` with properties `model_stats`, `model_coefs`, `ale`, `model_stats`, `boot_data`, and `params`.

Properties

model_stats `tibble` of bootstrapped results from `broom::glance()`. `NULL` if `model_stats` argument is `FALSE`. In general, only `broom::glance()` results that make sense when bootstrapped are included, such as `df` and `adj.r.squared`. Results that are incomparable across bootstrapped datasets (such as `aic`) are excluded. In addition, certain model performance measures are included; these are bootstrap-validated with the .632 correction (Efron & Tibshirani 1986) (NOT the .632+ correction):

- For regression (numeric prediction) models:
  - `mae`: mean absolute error (MAE)
  - `sa_mae`: standardized accuracy of the MAE referenced on the mean absolute deviation
  - `rmse`: root mean squared error (RMSE)
  - `sa_rmse`: standardized accuracy of the RMSE referenced on the standard deviation
- For binary or categorical classification (probability) models:
  - `auc`: area under the ROC curve

model_coefs A `tibble` of bootstrapped results from `broom::tidy()`. `NULL` if `model_coefs` argument is `FALSE`.

ale A list of bootstrapped ALE results using default `ALE()` settings unless if overridden with `ale_options`. `NULL` if `ale` argument is `FALSE`. Elements are:

```
* `single`: an `ALE` object of ALE calculations on the full dataset without bootstrapping.
* `boot`: a list of bootstrapped ALE data and statistics. This element is not an `ALE` object; it u
```

boot_data A `tibble` of bootstrap results. Each row represents a bootstrap iteration. `NULL` if `boot_data` argument is `FALSE`. The columns are:

```
* `it`: the specific bootstrap iteration from 0 to `boot_it` iterations. Iteration 0 is the result
* `row_idxs`: the row indexes for the bootstrapped sample for that iteration. To save space, the r
 * `model`: the model object trained on that iteration.
 * `ale`: the results of `ALE()` on that iteration.
 * `tidy`: the results of `broom::tidy(model)` on that iteration.
 * `stats`: the results of `broom::glance(model)` on that iteration.
* `perf`: performance measures on the entire dataset. These are the measures specified above for r
```

params Parameters used to calculate bootstrapped data. Most of these repeat the arguments passed to `ModelBoot()`. These are either the values provided by the user or used by default if the user did not change them but the following additional objects created internally are also provided:

```
* `y_cats`: same as `ALE@params$y_cats` (see documentation there).
* `y_type`: same as `ALE@params$y_type` (see documentation there).
* `model`: same as `ALE@params$model` (see documentation there).
* `data`: same as `ALE@params$data` (see documentation there).
```

Full-model bootstrapping

No modelling results, with or without ALE, should be considered reliable without appropriate validation. For ALE, both the trained model itself and the ALE that explains the trained model must be validated. ALE must be validated by bootstrapping. The trained model might be validated either by cross-validation or by bootstrapping. For ALE that explains trained models that have been developed by cross-validation, it is sufficient to bootstrap just the training data. That is what the `ALE` object does with its `boot_it` argument. However, unvalidated models must be validated by bootstrapping them along with the calculation of ALE; this is what the `ModelBoot` object does with its `boot_it` argument.

`ModelBoot()` carries out full-model bootstrapping to validate models. Specifically, it:

- Creates multiple bootstrap samples (default 100; the user can specify any number);

- Creates a model on each bootstrap sample;
- Calculates overall model statistics, variable coefficients, and ALE values for each model on each bootstrap sample;
- Calculates the mean, median, and lower and upper confidence intervals for each of those values across all bootstrap samples.

References

Okoli, Chitu. 2023. “Statistical Inference Using Machine Learning and Classical Techniques Based on Accumulated Local Effects (ALE).” arXiv. doi:10.48550/arXiv.2310.09877.<

Efron, Bradley, and Robert Tibshirani. "Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy." Statistical science (1986): 54-75. doi:10.1214/ss/1177013815

Examples

```
# attitude dataset
attitude

## ALE for generalized additive models (GAM)
## GAM is tweaked to work on the small dataset.
gam_attitude <- mgcv::gam(rating ~ complaints + privileges + s(learning) +
                            raises + s(critical) + advance,
                          data = attitude)
summary(gam_attitude)


# Full model bootstrapping
# Only 4 bootstrap iterations for a rapid example; default is 100
# Increase value of boot_it for more realistic results
mb_gam <- ModelBoot(
  gam_attitude,
  boot_it = 4
)

# If the model is not standard, supply model_call_string with 'data = boot_data'
# in the string instead of the actual dataset name (in addition to the actual dataset
# as the 'data' argument directly to the `ModelBoot` constructor).
mb_gam <- ModelBoot(
  gam_attitude,
  data = attitude,  # the actual dataset
  model_call_string = 'mgcv::gam(
    rating ~ complaints + privileges + s(learning) +
      raises + s(critical) + advance,
    data = boot_data  # required for model_call_string
  )',
  boot_it = 4
)

# Model statistics and coefficients
mb_gam@model_stats
```

```
mb_gam@model_coefs

# Plot ALE
plot(mb_gam)

# Retrieve ALE data
get(mb_gam, type = 'boot')    # bootstrapped
get(mb_gam, type = 'single')  # full (unbootstrapped) model
# See get.ALE() for other options
```

`plot.ALE` *plot method for* `ALE` *objects*

Description

This plot method simply calls the constructor for an `ALEPlots` object.

Arguments

`x`	ALE object.
`...`	Arguments passed to `ALEPlots()`

`plot.ALEPlots` *Plot method for ALEPlots object*

Description

Plot an `ALEPlots` object.

Arguments

`x`	An object of class `ALEPlots`.
`max_print`	integer(1). The maximum number of plots that may be printed at a time. 1D plots and 2D are printed on separate pages, so this maximum applies separately to each dimension of ALE plots, not to all dimensions combined.
`...`	Arguments to pass to `patchwork::wrap_plots()`

Value

Invisibly returns `x`.

`plot.ModelBoot` *plot method for* `ModelBoot` *objects*

Description

This plot method simply calls the constructor for an `ALEPlots` object.

Arguments

`x`	ModelBoot object.
`...`	Arguments passed to `ALEPlots()`

`print.ALE` *print Method for ALE object*

Description

Print an ALE object.

Arguments

`x`	An object of class `ALE`.
`...`	Additional arguments (currently not used).

Value

Invisibly returns `x`.

Examples

```
lm_cars <- stats::lm(mpg ~ ., mtcars)
ale_cars <- ALE(lm_cars, p_values = NULL)
print(ale_cars)
```

`print.ALEPlots` *Print method for ALEPlots object*

Description

Print an ALEPlots object by calling plot().

Arguments

`x`	An object of class `ALEPlots`.
`max_print`	See documentation for `plot.ALEPlots()`
`...`	Additional arguments (currently not used).

Value

Invisibly returns `x`.

`print.ModelBoot` *print method for ModelBoot object*

Description

Print a ModelBoot object.

Arguments

`x`	An object of class `ModelBoot`.
`...`	Additional arguments (currently not used).

Value

Invisibly returns `x`.

Examples

```
lm_cars <- stats::lm(mpg ~ wt + gear, mtcars)
mb <- ModelBoot(lm_cars, boot_it = 2, ale_p = NULL)
print(mb)
```

`resolve_x_cols` *Resolve x_cols and exclude_cols to their standardized format*

Description

Resolve `x_cols` and `exclude_cols` to their standardized format of `x_cols` to specify which 1D and 2D ALE elements are required. This specification is used throughout the ALE package. `x_cols` specifies the desired columns or interactions whereas `exclude_cols` optionally specifies any columns or interactions to remove from `x_cols`. The result is `x_cols` – `exclude_cols`, giving considerable flexibility in specifying the precise columns desired.

Usage

```
resolve_x_cols(x_cols, col_names, y_col, exclude_cols = NULL, silent = FALSE)
```

Arguments

`x_cols`	character, list, or formula. Columns and interactions requested in one of the special `x_cols` formats. `x_cols` variable names not found in `col_names` will error. See examples.
`col_names`	character. All the column names from a dataset. All values in `x_cols` must be contained among the values in `col_names`. For interaction terms in `x_cols`, e.g., `"a:b"`, the individual variable names must be contained in `col_names`, e.g, `c("a", "b")`.
`y_col`	character(1). The y outcome column. If found in any `x_cols` value, it will be silently removed.
`exclude_cols`	Same possible formats as `x_cols`. Columns and interactions to exclude from those requested in `x_cols`. `exclude_cols` values not found in `col_names` will be ignored with a message (which can be silenced with `silent`).
`silent`	logical(1). If `TRUE`, no message will be given; in particular, `x_cols` not found in `col_names` will be silently ignored. Default is `FALSE`. Regardless, warnings and errors are never silenced (e.g, invalid `x_cols` formats will still report errors).

Value

`x_cols` in canonical format, which is always a list with two elements, `d1` and `d2`. Each element is a character vector with each requested column for 1D ALE (`d1`) or 2D ALE interaction pair (`d2`). If either dimension is empty, its value is an empty character, `character()`.

See examples for details.

`x_cols` format options

The `x_cols` argument determines which predictor variables and interactions are included in the analysis. It supports multiple input formats:

- **Character vector**: Users can explicitly specify 1D terms and 2D ALE interactions, e.g., `c("a", "b", "a:b", "a:c")`.

- **Formula (~)**: Allows specifying variables and interactions in formula notation (e.g., `~ a + b + a:b`), which is automatically converted into a structured format. The outcome term is optional and will be ignored regardless. So, `~ a + b + a:b` produces results identical to `whatever ~ a + b + a:b`.
- **List format**:
  - The basic list format is a list of character vectors named `d1` for 1D ALE terms, `d2` for 2D interactions, or both. For example, `list(d1 = c("a", "b"), d2 = c("a:b", "a:c"))`
  - **Boolean selection for an entire dimension**:
    - `list(d1 = TRUE)` selects all available variables for 1D ALE, excluding `y_col`.
    - `list(d2 = TRUE)` selects all possible 2D interactions among all columns in `col_names`, excluding `y_col`.
  - A character vector of 1D terms only named `d2_all` may be used to include all 2D interactions that include the specified 1D terms. For example, specifying `list(d2_all = "a")` would select `c("a:b", "a:c", "a:d")`, etc. This is in addition to any terms requested in the `d1` or `d2` elements.
- **NULL (or unspecified)**: If `x_cols = NULL`, no variables are selected.

The function ensures all variables are valid and in `col_names`, providing informative messages unless `silent = TRUE`. And regardless of the specification format, the result will always be standardized in the format specified in the return value. Note that `y_col` is not removed if included in `x_cols`. However, a message alerts when it is included, in case it is a mistake.

Run examples for details.

Examples

```
## Sample data
set.seed(0)
df <- data.frame(
  y = runif(10),
  a = sample(letters[1:3], 10, replace = TRUE),
  b = rnorm(10),
  c = sample(1:5, 10, replace = TRUE)
)
col_names <- names(df)
y_col <- "y"  # Assume 'y' is the outcome variable


## Examples with just x_cols to show different formats for specifying x_cols
## (same format for exclude_cols)

# Character vector: Simple ALE with no interactions
resolve_x_cols(c("a", "b"), col_names, y_col)

# Character string: Select just one 1D element
resolve_x_cols("c", col_names, y_col)

# list of 1- and 2-length character vectors: specify precise 1D and 2D elements desired
resolve_x_cols(c('a:b', "c", 'c:a', "b"), col_names, y_col)
```

```
# Formula: Converts to a list of individual elements
resolve_x_cols(~ a + b, col_names, y_col)

# Formula with interactions (1D and 2D).
# This format is probably more convenient if you know precisely which terms you want.
# Note that the outcome on the left-hand-side is always silently ignored.
resolve_x_cols(whatever ~ a + b + a:b + c:b, col_names, y_col)

# List specifying d1 (1D ALE)
resolve_x_cols(list(d1 = c("a", "b")), col_names, y_col)

# List specifying d2 (2D ALE)
resolve_x_cols(list(d2 = 'a:b'), col_names, y_col)

# List specifying both d1 and d2
resolve_x_cols(list(d1 = c("a", "b"), d2 = 'a:b'), col_names, y_col)

# d1 as TRUE (select all columns except y_col)
resolve_x_cols(list(d1 = TRUE), col_names, y_col)

# d2 as TRUE (select all possible 2D interactions)
resolve_x_cols(list(d2 = TRUE), col_names, y_col)

# d2_all: Request all 2D interactions involving a specific variable
resolve_x_cols(list(d2_all = "a"), col_names, y_col)

# NULL: No variables selected
resolve_x_cols(NULL, col_names, y_col)



## Examples of how exclude_cols are removed from x_cols to obtain various desired results

# Exclude one column from a simple character vector
resolve_x_cols(
  x_cols = c("a", "b", "c"),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = "b"
)

# Exclude multiple columns
resolve_x_cols(
  x_cols = c("a", "b", "c"),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = c("a", "c")
)

# Exclude an interaction term from a formula input
resolve_x_cols(
  x_cols = ~ a + b + a:b,
  col_names = col_names,
```

```
  y_col = y_col,
  exclude_cols = ~ a:b
)

# Exclude all columns from x_cols
resolve_x_cols(
  x_cols = c("a", "b", "c"),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = c("a", "b", "c")
)

# Exclude non-existent columns (should be ignored)
resolve_x_cols(
  x_cols = c("a", "b"),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = "z"
)

# Exclude one column from a list-based input
resolve_x_cols(
  x_cols = list(d1 = c("a", "b"), d2 = c("a:b", "a:c")),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = list(d1 = "a")
)

# Exclude interactions only
resolve_x_cols(
  x_cols = list(d1 = c("a", "b", "c"), d2 = c("a:b", "a:c")),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = list(d2 = 'a:b')
)

# Exclude everything, including interactions
resolve_x_cols(
  x_cols = list(d1 = c("a", "b", "c"), d2 = c("a:b", "a:c")),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = list(d1 = c("a", "b", "c"), d2 = c("a:b", "a:c"))
)

# Exclude a column implicitly removed by y_col
resolve_x_cols(
  x_cols = c("y", "a", "b"),
  col_names = col_names,
  y_col = "y",
  exclude_cols = "a"
)

# Exclude entire 2D dimension from x_cols with d2 = TRUE
```

```
resolve_x_cols(
  x_cols = list(d1 = TRUE, d2 = c("a:b", "a:c")),
  col_names = col_names,
  y_col = y_col,
  exclude_cols = list(d1 = c("a"), d2 = TRUE)
)
```

`subset.ALEPlots` *subset method for ALEPlots object*

Description

Subset an `ALEPlots` object to produce another `ALEPlots` object only with the subsetted `x_cols` variables and interactions, as specified in the return value description.

See `get.ALE()` for explanation of parameters not described here.

Arguments

`x`	An object of class `ALEPlots`.
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`include_eff`	logical(1). `x_cols` and `exclude_cols` specify precisely which variables to include or exclude in the subset. However, multivariable plots like ALE effects plot are ambiguous because they cannot be subsetted to remove some existing variables. `include_eff = TRUE` (default) includes the ALE effects plot in the subset rather than dropping it, if it is available.

Value

An `ALEPlots` object reduced to cover only variables and interactions specified by `x_cols` and `exclude_cols`. This is different from `get.ALEPlots()`, which returns a list of `ggplot` objects and loses the special `ALEPlots` behaviour like plotting, printing, and summarizing multiple plots.

`summary.ALEPlots` *summary method for ALEPlots object*

Description

Present concise summary information about an `ALEPlots` object.

Arguments

`object`	An object of class `ALEPlots`.
`...`	Not used

Value

Summary string.

`var_cars` *Multi-variable transformation of the mtcars dataset.*

Description

This is a transformation of the `mtcars` dataset from R to produce a small dataset with each of the fundamental datatypes: logical, factor, ordered, integer, double, and character. Most of the transformations are obvious, but a few are noteworthy:

- The row names (the car model) are saved as a character vector.
- For the unordered factors, the country and continent of the car manufacturer are obtained based on the row names (model).
- For the ordered factor, gears 3, 4, and 5 are encoded as 'three', 'four', and 'five', respectively. The text labels make it explicit that the variable is ordinal, yet the number names make the order crystal clear.

Here is the adaptation of the original description of the `mtcars` dataset:

The data was extracted from the 1974 *Motor Trend* US magazine, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles (1973–74 models).

Usage

```
var_cars
```

Format

A tibble with 32 observations on 14 variables.

model `character`: Car model

mpg `double`: Miles/(US) gallon

cyl `integer`: Number of cylinders

disp `double`: Displacement (cu.in.)

hp `double`: Gross horsepower

drat `double`: Rear axle ratio

wt `double`: Weight (1000 lbs)

qsec `double`: 1/4 mile time

vs `logical`: Engine (0 = V-shaped, 1 = straight)

am `logical`: Transmission (0 = automatic, 1 = manual)

gear `ordered`: Number of forward gears

carb `integer`: Number of carburetors

country `factor`: Country of car manufacturer

continent `factor`: Continent of car manufacturer

Note

Henderson and Velleman (1981) comment in a footnote to Table 1: 'Hocking (original transcriber)'s noncrucial coding of the Mazda's rotary engine as a straight six-cylinder engine and the Porsche's flat engine as a V engine, as well as the inclusion of the diesel Mercedes 240D, have been retained to enable direct comparisons to be made with previous analyses.'

References

Henderson and Velleman (1981), Building multiple regression models interactively. *Biometrics*, **37**, 391–411.

Index