Package ‘ale’

February 14, 2024

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

Version 0.3.0

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 rewrites the original code from the 'ALEPlot' package for calculating ALE data and it completely reimplements the plotting of 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. <arXiv:2310.09877>. <doi:10.48550/arXiv.2310.09877>.

License GPL-2

Language en-ca

Encoding UTF-8

RoxygenNote 7.3.1

Suggests ALEPlot, knitr, mgcv, patchwork, readr, rmarkdown, testthat (>= 3.0.0)

VignetteBuilder knitr

Imports assertthat, broom, dplyr, ellipsis, furrr, future, ggplot2, ggpubr, glue, grDevices, insight, labeling, progressr, purrr, rlang, stats, stringr, tidyr, univariateML, yaImpute

Depends R (>= 3.5.0)

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

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

Config/testthat/edition 3

LazyData true

NeedsCompilation no

Author Chitu Okoli [aut, cre] (<https://orcid.org/0000-0001-5574-7572>),
Dan Apley [cph] (The current code for calculating ALE interaction
values is copied with few changes from Dan Apley's ALEPlot package.
We gratefully acknowledge his open-source contribution. However, he
was not directly involved in the development of this ale package.)

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

Repository CRAN

Date/Publication 2024-02-14 00:04:05 UTC

R topics documented:

---

ale — *Create and return ALE data, statistics, and plots*

Description

ale() is the central function that manages the creation of ALE data and plots for one-way ALE. For two-way interactions, see ale_ixn(). This function calls ale_core (a non-exported function) that manages the ALE data and plot creation in detail. For details, see the introductory vignette for this package or the details and examples below.

Usage

```
ale(
  data,
  model,
  x_cols = NULL,
  y_col = NULL,
  ...,
  parallel = parallel::detectCores(logical = FALSE) - 1,
  model_packages = as.character(NA),
  output = c("plots", "data", "stats", "conf_regions"),
  pred_fun = function(object, newdata, type = pred_type) {
     stats::predict(object =
    object, newdata = newdata, type = type)
 },
  pred_type = "response",
  p_values = NULL,
  p_alpha = c(0.01, 0.05),
  x_intervals = 100,
  boot_it = 0,
  seed = 0,
  boot_alpha = 0.05,
  boot_centre = "mean",
  relative_y = "median",
  y_type = NULL,
  median_band_pct = c(0.05, 0.5),
  rug_sample_size = 500,
  min_rug_per_interval = 1,
  ale_xs = NULL,
  ale_ns = NULL,
  compact_plots = FALSE,
  silent = FALSE
)
```

Arguments

`data`	dataframe. Dataset from which to create predictions for the ALE.
`model`	model object. Model for which ALE should be calculated. May be any kind of R object that can make predictions from data.
`x_cols`	character. Vector of column names from `data` for which one-way ALE data is to be calculated (that is, simple ALE without interactions). If not provided, ALE will be created for all columns in `data` except `y_col`.
`y_col`	character length 1. Name of the outcome target label (y) variable. If not provided, `ale()` will try to detect it automatically. For non-standard models, `y_col` should be provided. For survival models, set `y_col` to the name of the binary event column; in that case, `pred_type` should also be specified.
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`parallel`	non-negative integer length 1. Number of parallel threads (workers or tasks) for parallel execution of the function. See details.
`model_packages`	character. Character vector of names of packages that `model` depends on that might not be obvious. 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. If you get a strange error message that mentions something somewhere about 'future', try adding the package for your model to this vector, especially if you see such errors after the progress bars begin displaying (assuming you did not disable progress bars with `silent = TRUE`).
`output`	character in c('plots', 'data', 'stats', 'conf_regions'). Vector of types of results to return. 'plots' will return an ALE plot; 'data' will return the source ALE data; 'stats' will return ALE statistics. Each option must be listed to return the specified component. By default, all are returned.

`pred_fun, pred_type`	function,character length 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 and to determine the median band. If `NULL` (default), no p-values are calculated and `median_band_pct` is used to determine the median band. To calculate p-values, an object generated by the `create_p_funs()` function must be provided here. If `p_values` is set to 'auto', this `ale()` function will try to automatically create the p-values function; this only works with standard R model types. Any error message will be given if p-values cannot be generated. Any other input provided to this argument will result in an error. For more details about creating p-values, see documentation for `create_p_funs()`. Note that p-values will not be generated if 'stats' are not included as an option in the `output` argument.
`p_alpha`	numeric length 2 from 0 to 1. Alpha for "confidence interval" ranges for printing bands around the median for single-variable plots. These are the default values used if `p_values` are provided. If `p_values` are not provided, then `median_band_pct` is used instead. The inner band range will be the median value of y ± `p_alpha[2]` of the relevant ALE statistic (usually ALE range or normalized ALE range). For plots with a second outer band, its range will be the median ± `p_alpha[1]`. For example, in the ALE plots, for the default `p_alpha = c(0.01, 0.05)`, the inner band will be the median ± ALE minimum or maximum at p = 0.05 and the outer band will be the median ± ALE minimum or maximum at p = 0.01.
`x_intervals`	positive integer length 1. Maximum number of intervals on the x-axis for the ALE data for each column in `x_cols`. The number of intervals that the algorithm generates might eventually be fewer than what the user specifies if the data values for a given x value do not support that many intervals.
`boot_it`	non-negative integer length 1. Number of bootstrap iterations for the ALE values. If `boot_it = 0` (default), then ALE will be calculated on the entire dataset with no bootstrapping.
`seed`	integer length 1. Random seed. Supply this between runs to assure that identical random ALE data is generated each time
`boot_alpha`	numeric length 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 length 1 in c('mean', 'median'). When bootstrapping, the main estimate for `ale_y` is considered to be `boot_centre`. Regardless of the value specified here, both the mean and median will be available.
`relative_y`	character length 1 in c('median', 'mean', 'zero'). The ale_y values will be adjusted relative to this value. 'median' is the default. 'zero' will maintain the default of `ALEPlot::ALEPlot()`, which is not shifted.
`y_type`	character length 1. Datatype of the y (outcome) variable. Must be one of c('binary', 'numeric', 'multinomial', 'ordinal'). Normally determined automatically; only provide for complex non-standard models that require it.

`median_band_pct`
: numeric length 2 from 0 to 1. Alpha for "confidence interval" ranges for printing bands around the median for single-variable plots. These are the default values used if `p_values` are not provided. If `p_values` are provided, then `median_band_pct` is ignored. The inner band range will be the median value of y ± `median_band_pct[1]/2`. For plots with a second outer band, its range will be the median ± `median_band_pct[2]/2`. For example, for the default `median_band_pct = c(0.05, 0.5)`, the inner band will be the median ± 2.5% and the outer band will be the median ± 25%.

`rug_sample_size, min_rug_per_interval`
: single non-negative integer length 1. Rug plots are normally down-sampled otherwise they are too slow. `rug_sample_size` specifies the size of this sample. To prevent down-sampling, set to `Inf`. To suppress rug plots, set to 0. When down-sampling, the rug plots maintain representativeness of the data by guaranteeing that each of the `x_intervals` intervals will retain at least `min_rug_per_interval` elements; usually set to just 1 or 2.

`ale_xs, ale_ns`
: list of ale_x and ale_n vectors. If provided, these vectors will be used to set the intervals of the ALE x axis for each variable. By default (NULL), the function automatically calculates the ale_x intervals. `ale_xs` is normally used in advanced analyses where the ale_x intervals from a previous analysis are reused for subsequent analyses (for example, for full model bootstrapping; see the `model_bootstrap()` function).

`compact_plots`
: logical length 1, default `FALSE`. When `output` includes 'plots', the returned `ggplot` objects each include the environments of the plots. This lets the user modify the plots with all the flexibility of `ggplot`, but it can result in very large return objects (sometimes even hundreds of megabytes large). To compact the plots to their bare minimum, set `compact_plots = TRUE`. However, returned plots will not be easily modifiable, so this should only be used if you do not want to subsequently modify the plots.

`silent`
: logical length 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 enable progress bars.

Details

ale_core.R

Core functions for the ale package: ale, ale_ixn, and ale_core

Value

list with the following elements:

- `data`: a list whose elements, named by each requested x variable, are each a tibble with the following columns:
  - `ale_x`: the values of each of the ALE x intervals or categories.
  - `ale_n`: the number of rows of data in each `ale_x` interval or category.

  - ale_y: the ALE function value calculated for that interval or category. For bootstrapped ALE, this is the same as ale_y_mean by default or ale_y_median if the boot_centre = 'median' argument is specified. Regardless, both ale_y_mean and ale_y_median are returned as columns here.
  - ale_y_lo, ale_y_hi: the lower and upper confidence intervals, respectively, for the bootstrapped ale_y value. Note: regardless what options are requested in the output argument, this data element is always returned.

- stats: if stats are requested in the output argument (as is the default), returns a list. If not requested, returns NULL. The returned list provides ALE statistics of the data element duplicated and presented from various perspectives in the following elements:
  - by_term: a list named by each requested x variable, each of whose elements is a tibble with the following columns:
    - statistic: the ALE statistic specified in the row (see the by_statistic element below).
    - estimate: the bootstrapped mean or median of the statistic, depending on the boot_centre argument to the ale() function. Regardless, both mean and median are returned as columns here.
    - conf.low, conf.high: the lower and upper confidence intervals, respectively, for the bootstrapped estimate.
  - by_statistic: list named by each of the following ALE statistics: aled, aler_min, aler_max, naled, naler_min, naler_max. See vignette('ale-statistics') for details.
  - estimate: a tibble whose data consists of the estimate values from the by_term element above. The columns are term (the variable name) and the statistic for which the estimate is given: aled, aler_min, aler_max, naled, naler_min, naler_max.
  - effects_plot: a ggplot object which is the ALE effects plot for all the x variables.

- plots: if plots are requested in the output argument (as is the default), returns a list whose elements, named by each requested x variable, are each a ggplot object of the ALE y values plotted against the x variable intervals. If plots is not included in output, this element is NULL.

- conf_regions: if conf_regions are requested in the output argument (as is the default), returns a list. If not requested, returns NULL. The returned list provides summaries of the confidence regions of the relevant ALE statistics of the data element. The list has the following elements:
  - by_term: a list named by each requested x variable, each of whose elements is a tibble with the relevant data for the confidence regions. (See vignette('ale-statistics') for details about confidence regions.)
  - significant: a tibble that summarizes the by_term to only show confidence regions that are statistically significant. Its columns are those from by_term plus a term column to specify which x variable is indicated by the respective row.
  - sig_criterion: a length-one character vector that reports which values were used to determine statistical significance: if p_values was provided to the ale() function, it will be used; otherwise, median_band_pct will be used.

- Various values echoed from the original call to the ale() function, provided to document the key elements used to calculate the ALE data, statistics, and plots: y_col, x_cols, boot_it, seed, boot_alpha, boot_centre, relative_y, y_type, median_band_pct, rug_sample_size. These are either the values provided by the user or used by default if the user did not change them.

- y_summary: summary statistics of y values used for the ALE calculation. These statistics are based on the actual values of y_col unless if y_type is a probability or other value that is constrained in the [0, 1] range. In that case, y_summary is based on the predicted values of y_col by applying model to the data. y_summary is a named numeric vector. Most of the elements are the percentile of the y values. E.g., the '5%' element is the 5th percentile of y values. The following elements have special meanings:
  - The first element is named either p or q and its value is always 0. The value is not used; only the name of the element is meaningful. p means that the following special y_summary elements are based on the provided p_values object. q means that quantiles were calculated based on median_band_pct because p_values was not provided.
  - min, mean, max: the minimum, mean, and maximum y values, respectively. Note that the median is 50%, the 50th percentile.
  - med_lo_2, med_lo, med_hi, med_hi_2: med_lo and med_hi are the inner lower and upper confidence intervals of y values with respect to the median (50%); med_lo_2 and med_hi_2 are the outer confidence intervals. See the documentation for the p_alpha and median_band_pct arguments to understand how these are determined.

Custom predict function

The calculation of ALE requires modifying several values of the original data. Thus, ale() needs direct access to a predict function that work on 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. Most of the time, 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
  - 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 of numeric values as the prediction.

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

Note: survival models probably do not need a custom prediction function but y_col must be set to the name of the binary event column and pred_type must be set to the desired prediction type.

ALE statistics

For details about the ALE-based statistics (ALED, ALER, NALED, and NALER), see vignette('ale-statistics').

Parallel processing

Parallel processing using the {furrr} library is enabled by default. By default, it will use all the available physical CPU cores (minus the core being used for the current R session) with the setting parallel = parallel::detectCores(logical = FALSE) - 1. Note that only physical cores are used (not logical cores or "hyperthreading") because machine learning can only take advantage of the floating point processors on physical cores, which are absent from logical cores. Trying to use logical cores will not speed up processing and might actually slow it down with useless data transfer. If you will dedicate the entire computer to running this function (and you don't mind everything else becoming very slow while it runs), you may use all cores by setting parallel = parallel::detectCores(logical = FALSE). To disable parallel processing, set parallel = 0.

Progress bars

Progress bars are implemented with the {progressr} package, which lets the user fully control progress bars. **To disable progress bars, set silent = TRUE.** The first time a function is called in the {ale} package that requires progress bars, it checks if the user has activated the necessary {progressr} settings. If not, the {ale} package automatically enables {progressr} progress bars with the cli handler and prints a message notifying the user.

If you like the default progress bars and you want to make them permanent, then you can add the following lines of code to your .Rprofile configuration file and they will become your defaults for every R session; you will not see the message again:

```
progressr::handlers(global = TRUE)
progressr::handlers('cli')
```

For more details on formatting progress bars to your liking, see the introduction to the {progressr} package.

References

Okoli, Chitu. 2023. “Statistical Inference Using Machine Learning and Classical Techniques Based on Accumulated Local Effects (ALE).” arXiv. https://arxiv.org/abs/2310.09877.

Examples

```
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,
  data = diamonds_sample
)
summary(gam_diamonds)
```

```
# Simple ALE without bootstrapping
ale_gam_diamonds <- ale(
  diamonds_sample, gam_diamonds,
  parallel = 2  # CRAN limit (delete this line on your own computer)
)

# Plot the ALE data
ale_gam_diamonds$plots |>
  patchwork::wrap_plots()

# 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(
  diamonds_sample, gam_diamonds, boot_it = 100,
  parallel = 2  # CRAN limit (delete this line on your own computer)
)

# Bootstrapped ALEs print with confidence intervals
ale_gam_diamonds_boot$plots |>
  patchwork::wrap_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(
  diamonds_sample, gam_diamonds,
  pred_fun = custom_predict, pred_type = 'link',
  parallel = 2  # CRAN limit (delete this line on your own computer)
)

# Plot the ALE data
ale_gam_diamonds_custom$plots |>
  patchwork::wrap_plots()
```

ale_ixn *Create and return ALE interaction data, statistics, and plots*

Description

This is the central function that manages the creation of ALE data and plots for two-way ALE interactions. For simple one-way ALE, see `ale()`. See documentation there for functionality shared between both functions.

For details, see the introductory vignette for this package or the details and examples below.

For the plots, `n_y_quant` is the number of quantiles into which to divide the predicted variable (y). The middle quantiles are grouped specially:

- The middle quantile is the first confidence interval of `median_band_pct` (`median_band_pct[1]`) around the median. This middle quantile is special because it generally represents no meaningful interaction.
- The quantiles above and below the middle are extended from the borders of the middle quantile to the regular borders of the other quantiles.

There will always be an odd number of quantiles: the special middle quantile plus an equal number of quantiles on each side of it. If n_y_quant is even, then a middle quantile will be added to it. If n_y_quant is odd, then the number specified will be used, including the middle quantile.

Usage

```
ale_ixn(
  data,
  model,
  x1_cols = NULL,
  x2_cols = NULL,
  y_col = NULL,
  ...,
  parallel = parallel::detectCores(logical = FALSE) - 1,
  model_packages = as.character(NA),
  output = c("plots", "data"),
  pred_fun = function(object, newdata, type = pred_type) {
     stats::predict(object =
    object, newdata = newdata, type = type)
 },
  pred_type = "response",
  x_intervals = 100,
  relative_y = "median",
  y_type = NULL,
  median_band_pct = c(0.05, 0.5),
  rug_sample_size = 500,
  min_rug_per_interval = 1,
  ale_xs = NULL,
  n_x1_int = 20,
  n_x2_int = 20,
  n_y_quant = 10,
  compact_plots = FALSE,
  silent = FALSE
)
```

Arguments

`data`	See documentation for `ale()`
`model`	See documentation for `ale()`
`x1_cols, x2_cols`	character. Vectors of column names from `data` for which two-way interaction ALE data is to be calculated. ALE data will be calculated for each x1 column interacting with each x2 column. x1_cols can be of any standard datatype (logical, factor, or numeric) but x2_cols can only be numeric. If `ixn` is TRUE, then both values must be provided.
`y_col`	See documentation for `ale()`
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`parallel`	See documentation for `ale()`
`model_packages`	See documentation for `ale()`
`output`	See documentation for `ale()`
`pred_fun, pred_type`	See documentation for `ale()`
`x_intervals`	See documentation for `ale()`
`relative_y`	See documentation for `ale()`
`y_type`	See documentation for `ale()`
`median_band_pct`	See documentation for `ale()`
`rug_sample_size, min_rug_per_interval`	See documentation for `ale()`
`ale_xs`	See documentation for `ale()`
`n_x1_int, n_x2_int`	positive scalar integer. Number of intervals for the x1 or x2 axes respectively for interaction plot. These values are ignored if x1 or x2 are not numeric (i.e, if they are logical or factors).
`n_y_quant`	positive scalar integer. Number of intervals over which the range of y values is divided for the colour bands of the interaction plot. See details.
`compact_plots`	See documentation for `ale()`
`silent`	See documentation for `ale()`

Value

list of ALE interaction data tibbles and plots. The list has two levels of depth:

- The first level is named by the x1 variables.
- Within each x1 variable list, the second level is named by the x2 variables.
- Within each x1-x2 list element, the data or plot is returned as requested in the `output` argument.

Examples

```
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,
  data = diamonds_sample
)
summary(gam_diamonds)


# ALE two-way interactions
ale_ixn_gam_diamonds <- ale_ixn(
  diamonds_sample, gam_diamonds,
  parallel = 2  # CRAN limit (delete this line on your own computer)
)

# Print interaction plots
ale_ixn_gam_diamonds$plots |>
  # extract list of x1 ALE outputs
  purrr::walk(\(.x1) {
    # plot all x2 plots in each .x1 element
    patchwork::wrap_plots(.x1) |>
      print()
  })
```

---

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.

`create_p_funs` *Create a p-value functions object that can be used to generate p-values*

Description

Calculating p-values is not trivial for ALE statistics because ALE is non-parametric and model-agnostic. Because ALE is non-parametric (that is, it does not assume any particular distribution of data), the `ale` package generates p-values by calculating ALE for many random variables; this makes the procedure somewhat slow. For this reason, they are not calculated by default; they must be explicitly requested. Because the `ale` package is model-agnostic (that is, it works with any kind of R model), the `ale()` function 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 built-in 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 ale function 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 `create_p_funs()` 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.

Usage

```
create_p_funs(
  data,
  model,
  ...,
  parallel = parallel::detectCores(logical = FALSE) - 1,
  model_packages = as.character(NA),
  random_model_call_string = NULL,
  random_model_call_string_vars = character(),
  y_col = NULL,
  pred_fun = function(object, newdata, type = pred_type) {
     stats::predict(object =
    object, newdata = newdata, type = type)
 },
  pred_type = "response",
  rand_it = 1000,
  silent = FALSE,
  .testing_mode = FALSE
)
```

Arguments

`data`	See documentation for `ale()`
`model`	See documentation for `ale()`
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`parallel`	See documentation for `ale()`
`model_packages`	See documentation for `ale()`
`random_model_call_string`	character string. If NULL, `create_p_funs()` tries to automatically detect and construct the call for p-values. 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 the random variable. See details.

`random_model_call_string_vars`	See documentation for `model_call_string_vars` in `model_bootstrap()`; the operation is very similar.
`y_col`	See documentation for `ale()`
`pred_fun, pred_type`	See documentation for `ale()`.
`rand_it`	non-negative integer length 1. Number of times that the model should be retrained with a new random variable. The default of 1000 should give reasonably stable p-values. It can be reduced as low as 100 for faster test runs.
`silent`	See documentation for `ale()`
`.testing_mode`	logical. Internal use only.

Value

The return value is a list of class `c('p_funs', 'ale', 'list')` with an `ale_version` attribute whose value is the version of the `ale` package used to create the object. See examples for an illustration of how to inspect this list. Its elements are:

- `value_to_p`: a list of functions named for each each available ALE statistic. Each function signature is `function(x)` where x is a numeric. The function returns the p-value (minimum 0; maximum 1) for the respective statistic based on the random variable analysis. For an input x that returns p, its interpretation is that p% of random variables obtained the same or higher statistic value. For example, to get the p-value of a NALED of 4.2, enter `p_funs$value_to_p(4.2)`. A return value of 0.03 means that only 3% of random variables obtained a NALED greater than or equal to 4.2.
- `p_to_random_value`: a list of functions named for each each available ALE statistic. These are the inverse functions of `value_to_p`. The signature is `function(p)` where p is a numeric from 0 to 1. The function returns the numeric value of the random variable statistic that would yield the provided p-value. For an input p that returns x, its interpretation is that p% of random variables obtained the same or higher statistic value. For example, to get the random variable ALED for the 0.05 p-value, enter `p_funs$p_to_random_value(0.05)`. A return value of 102 means that only 5% of random variables obtained an ALED greater than or equal to 102.
- `rand_stats`: a tibble whose rows are each of the `rand_it` iterations of the random variable analysis and whose columns are the ALE statistics obtained for each random variable.
- `residuals`: the actual `y_col` values from `data` minus the predicted values from the `model` (without random variables) on the `data`. `residual_distribution`: the closest estimated distribution for the `residuals` as determined by `univariateML::rml()`. This is the distribution used to generate all the random variables.

Approach to calculating p-values

The `ale` package takes a literal frequentist approach to the calculation of 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 (from `stats::ecdf()`) is used to create a function to determine p-values according to the distribution of the random variables' ALE statistics.

References

Okoli, Chitu. 2023. "Statistical Inference Using Machine Learning and Classical Techniques Based on Accumulated Local Effects (ALE)." arXiv. https://arxiv.org/abs/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,
  data = diamonds_sample
)
summary(gam_diamonds)

# Create p-value functions
pf_diamonds <- create_p_funs(
  diamonds_sample,
  gam_diamonds,
  # only 100 iterations for a quick demo; but usually should remain at 1000
  rand_it = 100,
)

# Examine the structure of the returned object
str(pf_diamonds)
# In RStudio: View(pf_diamonds)

# Calculate ALEs with p-values
ale_gam_diamonds <- ale(
  diamonds_sample,
  gam_diamonds,
  p_values = pf_diamonds
)

# Plot the ALE data. The horizontal bands in the plots use the p-values.
```

```
ale_gam_diamonds$plots |>
  patchwork::wrap_plots()


# 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.
pf_diamonds <- create_p_funs(
  diamonds_sample,
  gam_diamonds,
  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,
)
```

model_bootstrap *model_bootstrap.R*

Description

Execute full model bootstrapping with ALE calculation on each bootstrap run

Usage

```
model_bootstrap(
  data,
  model,
  ...,
  model_call_string = NULL,
  model_call_string_vars = character(),
  parallel = parallel::detectCores(logical = FALSE) - 1,
  model_packages = as.character(NA),
  boot_it = 100,
  seed = 0,
  boot_alpha = 0.05,
  boot_centre = "mean",
  output = c("ale", "model_stats", "model_coefs"),
  ale_options = list(),
  tidy_options = list(),
  glance_options = list(),
  compact_plots = FALSE,
  silent = FALSE
)
```

Arguments

`data`	dataframe. Dataset that will be bootstrapped.
`model`	See documentation for `ale()`
`...`	not used. Inserted to require explicit naming of subsequent arguments.
`model_call_string`	character string. If NULL, `model_bootstrap()` 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. Character vector of 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 will produce an error. If parallelization is disabled with `parallel = 0`, then this is not a concern.
`parallel`	See documentation for `ale()`
`model_packages`	See documentation for `ale()`
`boot_it`	integer from 0 to Inf. Number of bootstrap iterations. If boot_it = 0, then the model is run as normal once on the full `data` with no bootstrapping.
`seed`	integer. Random seed. Supply this between runs to assure identical bootstrap samples are generated each time on the same data.
`boot_alpha`	numeric. The confidence level for the bootstrap confidence intervals is 1 - boot_alpha. For example, the default 0.05 will give a 95% confidence interval, that is, from the 2.5% to the 97.5% percentile.
`boot_centre`	See See documentation for `ale()`
`output`	character vector. Which types of bootstraps to calculate and return: • 'ale': Calculate and return bootstrapped ALE data and plot. • 'model_stats': Calculate and return bootstrapped overall model statistics. • 'model_coefs': Calculate and return bootstrapped model coefficients. • 'boot_data': Return full data for all bootstrap iterations. This data will always be calculated because it is needed for the bootstrap averages. By default, it is not returned except if included in this `output` argument.
`ale_options, tidy_options, glance_options`	list of named arguments. Arguments to pass to the `ale()`, `broom::tidy()`, or `broom::glance()` functions, respectively, beyond (or overriding) the defaults. In particular, to obtain p-values for ALE statistics, see the details.
`compact_plots`	See documentation for `ale()`
`silent`	See documentation for `ale()`

Details

No modelling results, with or without ALE, should be considered reliable without being bootstrapped. For large datasets, normally the model provided to `ale()` is the final deployment model that has been validated and evaluated on training and testing on subsets; that is why `ale()` is calculated on the full dataset. However, when a dataset is too small to be subdivided into training and test sets for a standard machine learning process, then the entire model should be bootstrapped. That is, multiple models should be trained, one on each bootstrap sample. The reliable results are the average results of all the bootstrap models, however many there are. For details, see the vignette on small datasets or the details and examples below.

`model_bootstrap()` automatically carries out full-model bootstrapping suitable for small datasets. Specifically, it:

- Creates multiple bootstrap samples (default 100; the user can specify any number);
- Creates a model on each bootstrap sample;
- Calculates model overall 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.

P-values The `broom::tidy()` summary statistics will provide p-values as normal, but the situation is somewhat complicated with p-values for ALE statistics. The challenge is that the procedure for obtaining their p-values is very slow: it involves retraining the model 1000 times. Thus, it is not efficient to calculate p-values on every execution of `model_bootstrap()`. Although the `ale()` function provides an 'auto' option for creating p-values, that option is disabled in `model_bootstrap()` because it would be far too slow: it would involve retraining the model 1000 times the number of bootstrap iterations. Rather, you must first create a p-values function object using the procedure described in `help(create_p_funs)`. If the name of your p-values object is `p_funs`, you can then request p-values each time you run `model_bootstrap()` by passing it the argument `ale_options = list(p_values = p_funs)`.

Value

list with tibbles of the following elements (depending on values requested in the `output` argument:

- model_stats: bootstrapped results from `broom::glance()`
- model_coefs: bootstrapped results from `broom::tidy()`
- ale: bootstrapped ALE results
  - data: ALE data (see `ale()` for details about the format)
  - stats: ALE statistics. The same data is duplicated with different views that might be variously useful. The column
    - by_term: statistic, estimate, conf.low, median, mean, conf.high. ("term" means variable name.) The column names are compatible with the `broom` package. The confidence intervals are based on the `ale()` function defaults; they can be changed with the `ale_options` argument. The estimate is the median or the mean, depending on the `boot_centre` argument.
    - by_statistic: term, estimate, conf.low, median, mean, conf.high.

        * estimate: term, then one column per statistic Provided with the default estimate. This view does not present confidence intervals.
    - plots: ALE plots (see `ale()` for details about the format)
- boot_data: full bootstrap data (not returned by default)
- other values: the `boot_it`, `seed`, `boot_alpha`, and `boot_centre` arguments that were originally passed are returned for reference.

References

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

Examples

```
# attitude dataset
attitude

## ALE for general 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 <- model_bootstrap(
  attitude,
  gam_attitude,
  boot_it = 4,
  parallel = 2  # CRAN limit (delete this line on your own computer)
)

# If the model is not standard, supply model_call_string with
# 'data = boot_data' in the string (not as a direct argument to [model_bootstrap()])
mb_gam <- model_bootstrap(
  attitude,
  gam_attitude,
  model_call_string = 'mgcv::gam(
    rating ~ complaints + privileges + s(learning) +
      raises + s(critical) + advance,
    data = boot_data
  )',
  boot_it = 4,
  parallel = 2  # CRAN limit (delete this line on your own computer)
)

# Model statistics and coefficients
```

```
mb_gam$model_stats
mb_gam$model_coefs

# Plot ALE
mb_gam$ale$plots |>
  patchwork::wrap_plots()
```

`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, and double. Most of the transformations are obvious, but two are noteworthy:

- For the unordered factor, the country of the car manufacturer is obtained based on the row names of `mtcars`. This `var_cars` version does not have row names.
- 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 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 12 variables.

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

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