Package 'MLwrap'

May 7, 2026

Title Machine Learning Modelling for Everyone

Version 0.3.0

Description A minimal library specifically designed to make the estimation of Machine Learning (ML) techniques as easy and accessible as possible, particularly within the framework of the Knowledge Discovery in Databases (KDD) process in data mining. The package provides essential tools to structure and execute each stage of a predictive or classification modeling workflow, aligning closely with the fundamental steps of the KDD methodology, from data selection and preparation, through model building and tuning, to the interpretation and evaluation of results using Sensitivity Analysis. The 'MLwrap' workflow is organized into four core steps; preprocessing(), build_model(), fine_tuning(), and sensitivity_analysis(). It also includes global and pairwise interaction analysis based on Friedman's H-statistic to support a more detailed interpretation of complex feature relationships.These steps correspond, respectively, to data preparation and transformation, model construction, hyperparameter optimization, and sensitivity analysis. The user can access comprehensive model evaluation results including fit assessment metrics, plots, predictions, and performance diagnostics for ML models implemented through 'Neural Networks', 'Random Forest', 'XGBoost' (Extreme Gradient Boosting), and 'Support Vector Machines' (SVM) algorithms. By streamlining these phases, 'MLwrap' aims to simplify the implementation of ML techniques, allowing analysts and data scientists to focus on extracting actionable insights and meaningful patterns from large datasets, in line with the objectives of the KDD process.

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1.0)

Imports R6, tidyr, magrittr, dials, parsnip, recipes, rsample, tune, workflows, yardstick, glue, innsight, fastshap, DiagrammeR, ggbeeswarm, ggplot2, sensitivity, dplyr, rlang, tibble, patchwork, cli, scales

Suggests testthat (>= 3.0.0), torch, brulee, ranger, kernlab, xgboost

Config/testthat/edition 3

URL https://github.com/AlbertSesePsy/MLwrap

BugReports https://github.com/AlbertSesePsy/MLwrap/issues

LazyData true

NeedsCompilation no

Author Javier Martínez García [aut] (ORCID:
<https://orcid.org/0009-0007-7861-5274>),
Juan José Montaño Moreno [ctb] (ORCID:
<https://orcid.org/0000-0002-1116-1964>),
Albert Sesé [cre, ctb] (ORCID: <https://orcid.org/0000-0003-3771-1749>)

Maintainer Albert Sesé <albert.sese@uib.es>

Repository CRAN

Date/Publication 2025-12-15 08:30:02 UTC

Contents

`build_model` *Create ML Model*

Description

The function **build_model()** is designed to construct and attach a ML model to an existing analysis object,which contains the preprocessed dataset generated in the previous step using the preprocessing() function. Based on the specified model type and optional hyperparameters, it supports several popular algorithms—including **Neural Network**, **Random Forest**, **XGBOOST**, and **SVM** (James et al., 2021)— by initializing the corresponding hyperparameter class, updating the analysis object with these settings, and invoking the appropriate model creation function. For SVM models, it further distinguishes between kernel types (rbf, polynomial, linear) to ensure the correct implementation. The function also updates the analysis object with the model name, the fitted model, and the current processing stage before returning the enriched object, thereby streamlining the workflow for subsequent training, evaluation, or prediction steps. This modular approach facilitates flexible and reproducible ML pipelines by encapsulating both the model and its configuration within a single structured object.

Usage

```
build_model(analysis_object, model_name, hyperparameters = NULL)
```

Arguments

`analysis_object`
: analysis_object created from preprocessing function.

`model_name`
: Name of the ML Model. A string of the model name: "Neural Network", "Random Forest", "SVM" or "XGBOOST".

`hyperparameters`
: Hyperparameters of the ML model. List containing the name of the hyperparameter and its value or range of values.

Value

An updated analysis_object containing the fitted machine learning model, the model name, the specified hyperparameters, and the current processing stage. This enriched object retains all previously stored information from the preprocessing step and incorporates the results of the model-building process, ensuring a coherent and reproducible workflow for subsequent training, evaluation, or prediction tasks.

Hyperparameters

Neural Network:

Parsnip model using **brulee** engine. Hyperparameters:

- **hidden_units**: Number of Hidden Neurons. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(5, 20).
- **activation**: Activation Function. A vector with any of ("relu", "sigmoid", "tanh") or NULL for default values c("relu", "sigmoid", "tanh").
- **learn_rate**: Learning Rate. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(-3, -1) in log10 scale.

Random Forest:

Parsnip model using **ranger** engine. Hyperparameters:

- **trees**: Number of Trees. A single value, a vector with range values `c(min_val, max_val)`. Default range c(100, 300).
- **mtry**: Number of variables randomly selected as candidates at each split. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(3, 8).
- **min_n**: Minimum Number of samples to split at each node. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(5, 25).

XGBOOST:

Parsnip model using **xgboost** engine. Hyperparameters:

- **trees**: Number of Trees. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(100, 300).
- **mtry**: Number of variables randomly selected as candidates at each split. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(3, 8).
- **min_n**: Minimum Number of samples to split at each node. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(5, 25).
- **tree_depth**: Maximum tree depth. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(3, 8).
- **learn_rate**: Learning Rate. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(-3, -1) in log10 scale.
- **loss_reduction**: Minimum loss reduction required to make a further partition on a leaf node. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(-3, 1.5) in log10 scale.

SVM:

Parsnip model using **kernlab** engine. Hyperparameters:

- **cost**: Penalty parameter that regulates model complexity and misclassification tolerance. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(-3, 3) in log2 scale.
- **margin**: Distance between the separating hyperplane and the nearest data points. A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(0, 0.2).
- **type**: Kernel to be used. A single value from ("linear", "rbf", "polynomial"). Default: "linear".

- **rbf_sigma**: A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(-5, 0) in log10 scale.
- **degree**: Polynomial Degree (polynomial kernel only). A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(1, 3).
- **scale_factor**: Scaling coefficient applied to inputs. (polynomial kernel only) A single value, a vector with range values `c(min_val, max_val)` or NULL for default range c(-5, -1) in log10 scale.

References

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An Introduction to Statistical Learning: with Applications in R (2nd ed.)*. Springer. https://doi.org/10.1007/978-1-0716-1418-1

Examples

```
# Example 1: Random Forest for regression task

library(MLwrap)

data(sim_data) # sim_data is a simulated dataset with psychological variables

wrap_object <- preprocessing(
     df = sim_data,
     formula = psych_well ~ depression + emot_intel + resilience + life_sat,
     task = "regression"
     )

wrap_object <- build_model(
               analysis_object = wrap_object,
               model_name = "Random Forest",
               hyperparameters = list(
                                 mtry = 2,
                                 trees = 10
                                 )
                           )
# It is safe to reuse the same object name (e.g., wrap_object, or whatever)
# step by step, as all previous results and information are retained within
# the updated analysis object.

# Example 2: SVM for classification task

data(sim_data) # sim_data is a simulated dataset with psychological variables

wrap_object <- preprocessing(
         df = sim_data,
         formula = psych_well_bin ~ depression + emot_intel + resilience + life_sat,
         task = "classification"
         )

wrap_object <- build_model(
               analysis_object = wrap_object,
               model_name = "SVM",
```

```
                hyperparameters = list(
                                  type = "rbf",
                                  cost = 1,
                                  margin = 0.1,
                                  rbf_sigma = 0.05
                                  )
                              )
```

`fine_tuning` *Fine Tune ML Model*

Description

The **fine_tuning()** function performs automated hyperparameter optimization for ML workflows encapsulated within an AnalysisObject. It supports two tuning strategies: **Bayesian Optimization** (with cross-validation) and **Grid Search Cross-Validation**, allowing the user to specify evaluation metrics and whether to visualize tuning results. The function first validates arguments and updates the workflow and metric settings within the AnalysisObject. If hyperparameter tuning is enabled, it executes the selected tuning procedure, identifies the best hyperparameter configuration based on the specified metrics, and updates the workflow accordingly. For neural network models, it also manages the creation and integration of new model instances and provides additional visualization of training dynamics. Finally, the function fits the optimized model to the training data and updates the AnalysisObject, ensuring a reproducible and efficient model selection process (Bartz et al., 2023).

Usage

```
fine_tuning(analysis_object, tuner, metrics = NULL)
```

Arguments

`analysis_object`	analysis_object created from build_model function.
`tuner`	Name of the Hyperparameter Tuner. A string of the tuner name: "Bayesian Optimization" or "Grid Search CV".
`metrics`	Metric used for Model Selection. A string of the name of metric (see Metrics). By default either "rmse" (regression) or "roc_auc" (classification).

Value

An updated analysis_object containing the fitted model with optimized hyperparameters, the tuning results, and all relevant workflow modifications. This object includes the final trained model, the best hyperparameter configuration, tuning diagnostics, and, if applicable, plots of the tuning process. It can be used for further model evaluation, prediction, or downstream analysis within the package workflow.

Tuners

Bayesian Optimization (with cross-validation):

- Number of Folds: 5
- Initial data points: 20
- Maximum number of iterations: 25
- Convergence after 5 iterations without improvement
- Train / Test : 0.75 / 0.25

Grid Search CV:

- Number of Folds: 5
- Maximum levels per hyperparameter: 10
- Train / Test : 0.75 / 0.25

Metrics

Regression Metrics:

- rmse
- mae
- mpe
- mape
- ccc
- smape
- rpiq
- rsq

Classification Metrics:

- accuracy
- bal_accuracy
- recall
- sensitivity
- specificity
- kap
- f_meas
- mcc
- j_index
- detection_prevalence
- roc_auc
- pr_auc
- gain_capture
- brier_class
- roc_aunp

References

Bartz, E., Bartz-Beielstein, T., Zaefferer, M., & Mersmann, O. (2023). *Hyperparameter tuner for Machine and Deep Learning with R. A Practical Guide*. Springer. doi:10.1007/9789811951701

Examples

```
# Fine tuning function applied to a regression task using Random Forest

wrap_object <- preprocessing(
           df = sim_data[1:500 ,],
           formula = psych_well ~ depression + life_sat,
           task = "regression"
           )
wrap_object <- build_model(
               analysis_object = wrap_object,
               model_name = "Random Forest",
               hyperparameters = list(
                     mtry = 2,
                     trees = 3
                     )
                 )
set.seed(123) # For reproducibility
wrap_object <- fine_tuning(wrap_object,
                tuner = "Grid Search CV",
                metrics = c("rmse")
                )
```

`plot_ale` *Plot Accumulated Local Effects (ALE)*

Description

The **plot_ale()** function computes and visualizes **Accumulated Local Effects (ALE)** for a selected feature, following the approach described in *Interpretable Machine Learning* by Christoph Molnar. ALE plots quantify how changes in a feature locally influence model predictions, offering a robust alternative to Partial Dependence Plots (PDP) by avoiding extrapolation and handling correlated predictors more reliably.

Usage

```
plot_ale(
  analysis_object,
  feature,
  group = NULL,
  grid.size = 20,
  use_test = FALSE,
  plot = TRUE
)
```

Arguments

analysis_object	A fitted wrap_object with model results or previously computed ALE values.
feature	Character. Name of the continuous feature for which ALE should be computed.
group	Optional character. A grouping variable to compute grouped ALE curves.
grid.size	Integer. Number of intervals to partition the feature domain (default = 20).
use_test	Logical. If TRUE, ALE is computed using the test set (default = FALSE).
plot	Logical. If TRUE, displays the ALE plot and returns wrap_object; if FALSE, returns the ggplot object without modifying the object.

Value

If plot = TRUE, returns the updated wrap_object and prints the ALE plot. If plot = FALSE, returns a ggplot object containing the ALE visualization.

References

Molnar, C. (2022). *Interpretable Machine Learning*.
https://christophm.github.io/interpretable-ml-book/

See Also

sensitivity_analysis

Examples

```
# After fitting a model with fine_tuning(wrap_object):
# plot_ale(wrap_object, feature = "age")
```

---

plot_calibration_curve

Plotting Calibration Curve

Description

The **plot_calibration_curve()** function generates calibration plots for binary classification models evaluating the agreement between predicted probabilities and observed class frequencies in binned prediction intervals. Implements reliability diagrams comparing empirical success rates within each probability bin against the predicted probability levels, identifying systematic calibration errors including overconfidence (predicted probabilities exceed observed frequencies) and underconfidence patterns across prediction ranges.

Usage

```
plot_calibration_curve(analysis_object)
```

Arguments

analysis_object
: Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

Examples

```
# Note: For obtaining the calibration curve plot the user needs to
# complete till fine_tuning( ) function of the MLwrap pipeline and
# only with binary outcome.

wrap_object <- preprocessing(df = sim_data[1:300 ,],
                             formula = psych_well_bin ~ depression + resilience,
                             task = "classification")
wrap_object <- build_model(wrap_object, "Random Forest",
                           hyperparameters = list(mtry = 2, trees = 5))
set.seed(123) # For reproducibility
wrap_object <- fine_tuning(wrap_object, "Grid Search CV")

# And then, you can obtain the calibration curve plot.

plot_calibration_curve(wrap_object)
```

---

plot_confusion_matrix *Plotting Confusion Matrix*

Description

The **plot_confusion_matrix()** function generates confusion matrices from classification predictions displaying the contingency table of true class labels versus predicted class labels. Visualizes true positives, true negatives, false positives, and false negatives for both training and test sets, enabling computation of derived performance metrics (sensitivity, specificity, precision, F1-score) and identification of specific class pair misclassification patterns.

Usage

```
plot_confusion_matrix(analysis_object)
```

Arguments

analysis_object
: Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

plot_calibration_curve

Examples

```
# Note: For obtaining confusion matrix plot the user needs to
# complete till fine_tuning( ) function of the MLwrap pipeline and
# only with categorical outcome.
# See the full pipeline example under plot_calibration_curve()
# Final call signature:
# plot_confusion_matrix(wrap_object)
```

---

plot_distribution_by_class

Plotting Output Distribution By Class

Description

The **plot_distribution_by_class()** function visualizes kernel density estimates or histograms of predicted probability distributions stratified by true class labels. Enables assessment of class separability through probability overlap quantification and identification of prediction probability ranges where different classes exhibit substantial overlap, indicating classification ambiguity regions.

Usage

```
plot_distribution_by_class(analysis_object)
```

Arguments

analysis_object
: Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

plot_calibration_curve

Examples

```
# Note: For obtaining the distribution by class plot the user needs to
# complete till fine_tuning( ) function of the MLwrap pipeline
# and only with categorical outcome.
# See the full pipeline example under plot_calibration_curve()
# Final call signature:
# plot_distribution_by_class(wrap_object)
```

plot_gain_curve *Plotting Gain Curve*

Description

The **plot_gain_curve()** plots cumulative gain as a function of sorted population percentile when observations are ranked by descending predicted probability. For each percentile threshold, calculates the ratio of positive class proportion in the top-ranked subset relative to overall positive class proportion, quantifying model's efficiency in concentrating target cases at the top of rankings.

Usage

```
plot_gain_curve(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

plot_calibration_curve

Examples

```
# Note: For obtaining the gain curve plot the user needs to complete till
# fine_tuning( ) function of the MLwrap pipeline and only with categorical
# outcome.
# See the full pipeline example under plot_calibration_curve()
# Final call signature:
# plot_gain_curve(wrap_object)
```

plot_graph_nn *Plot Neural Network Architecture*

Description

Renders a directed acyclic graph representation of Neural Network architecture showing layer stacking order, layer-specific dimensions (neurons per layer), activation functions applied at each layer, and optimized hyperparameter values (learning rate, batch size, dropout rates, regularization coefficients) obtained from hyperparameter tuning procedures.

Usage

```
plot_graph_nn(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

table_best_hyperparameters

Examples

```
# Note: For obtaining the Neural Network architecture graph plot the user
# needs to complete till the fine_tuning( ) function of the MLwrap pipeline.
# See the full pipeline example under table_best_hyperparameters()
# (Neural Network engine required)
# Final call signature:
# plot_graph_nn(wrap_object)
```

---

plot_integrated_gradients

Plotting Integrated Gradients Plots

Description

The **plot_integrated_gradients()** function implements interpretability visualizations of integrated gradient attributions measuring feature importance through accumulated gradients along the interpolation path from baseline (zero vector) to observed input. Provides four visualization modalities: mean absolute attributions (bar plots), directional effects showing positive and negative contribution patterns (directional plots), distributional properties of attributions across instances (box plots), and individual-level attribution contributions (swarm plots).

Usage

```
plot_integrated_gradients(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "Integrated Gradients")'.
show_table	Boolean. Whether to print Integrated Gradients summarized results table.

Value

analysis_object

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the Integrated Gradients plot the user needs to
# complete till sensitivity_analysis( ) function of the MLwrap pipeline
# using the Integrated Gradients method.
# See the full pipeline example under sensitivity_analysis()
# (Requires sensitivity_analysis(methods = "Integrated Gradients"))
# Final call signature:
# plot_integrated_gradients(wrap_object)
```

---

plot_lift_curve *Plotting Lift Curve*

Description

The **plot_lift_curve()** function plots lift factor as a function of population percentile when observations are ranked by descending predicted probability. The lift factor quantifies model's ranking efficiency relative to random ordering baseline at each population cumulative segment, showing how much better model selection performs compared to random case selection.

Usage

```
plot_lift_curve(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

plot_calibration_curve

Examples

```
# Note: For obtaining the lift curve plot the user needs to complete till
# fine_tuning( ) function of the MLwrap pipeline and only with categorical
# outcome.
# See the full pipeline example under plot_calibration_curve()
# Final call signature:
# plot_lift_curve(wrap_object)
```

`plot_loss_curve` *Plot Neural Network Loss Curve*

Description

Displays training loss trajectory computed on the validation set across training epochs. Enables visual diagnosis of convergence dynamics, identification of appropriate early stopping points, detection of overfitting patterns (where validation loss increases while training loss decreases), and assessment of optimization stability throughout the training process.

Usage

```
plot_loss_curve(analysis_object)
```

Arguments

`analysis_object`
: Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

`table_best_hyperparameters`

Examples

```
# Note: For obtaining the loss curve plot the user needs to
# complete till the fine_tuning( ) function of the MLwrap pipeline.
# See the full pipeline example under table_best_hyperparameters()
# (Neural Network engine required)
# Final call signature:
# plot_loss_curve(wrap_object)
```

plot_olden *Plotting Olden Values Barplot*

Description

The **plot_olden()** function visualizes Olden sensitivity values computed from products of input-to-hidden layer connection weights and hidden-to-output layer connection weights for each feature. Provides relative feature importance rankings specific to feedforward Neural Networks based on synaptic weight magnitude and directionality analysis across network layers.

Usage

```
plot_olden(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "Olden")'.
show_table	Boolean. Whether to print Olden results table.

Value

analysis_object

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the Olden plot the user needs to complete till
# sensitivity_analysis( ) function of the MLwrap pipeline using the Olden
# method.
# See the full pipeline example under sensitivity_analysis()
# (Requires sensitivity_analysis(methods = "Olden"))
# Final call signature:
# plot_olden(wrap_object)
```

plot_pdp *Plot Partial Dependence (PDP)*

Description

The **plot_pdp()** function computes and visualizes **Partial Dependence Plots (PDP)** for a selected feature, following the methodology described in *Interpretable Machine Learning* by Christoph Molnar. PDPs show the average effect of a feature on model predictions by marginalizing over the distribution of all other features. Optionally, Individual Conditional Expectation (ICE) curves can be added to visualize heterogeneous effects.

Usage

```
plot_pdp(
  analysis_object,
  feature,
  group_by = NULL,
  grid_size = 25,
  show_ice = TRUE,
  ice_n = 50,
  pdp_line_size = 1.1,
  use_test = FALSE,
  plot = TRUE
)
```

Arguments

analysis_object	A fitted `wrap_object` with model results or previously computed PDP values.
feature	Character. The continuous feature for which the PDP should be computed.
group_by	Optional character. A variable used to produce grouped PDP curves.
grid_size	Integer. Number of points used to evaluate the PDP (default = 25).
show_ice	Logical. Whether to overlay ICE curves (default = TRUE).
ice_n	Integer. Number of ICE curves to sample if `show_ice = TRUE` (default = 50).
pdp_line_size	Numeric. Line width for the PDP curve (default = 1.1).
use_test	Logical. Compute PDP using the test set instead of the training set (default = FALSE).
plot	Logical. If TRUE, prints the PDP plot and returns `wrap_object`; if FALSE, returns the ggplot object without modifying the object.

Value

If `plot = TRUE`, returns the updated `wrap_object` and prints the PDP plot. If `plot = FALSE`, returns a ggplot object containing the PDP (and optionally ICE) visualization.

References

Molnar, C. (2022). *Interpretable Machine Learning*.
https://christophm.github.io/interpretable-ml-book/

See Also

sensitivity_analysis

Examples

```
# After fitting model with fine_tuning(wrap_object):
# plot_pdp(wrap_object, feature = "age")
```

---

plot_pfi *Plotting Permutation Feature Importance Barplot*

Description

The **plot_pfi()** function generates feature importance estimates via Permutation Feature Importance measuring performance degradation when each feature's values are randomly permuted while holding all other features constant. Provides model-agnostic importance ranking independent of feature-target correlation patterns, capturing both linear and non-linear predictive contributions to model performance.

Usage

```
plot_pfi(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "PFI")'.
show_table	Boolean. Whether to print PFI results table.

Value

analysis_object

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the PFI plot results the user needs to complete till
# sensitivity_analysis( ) function of the MLwrap pipeline using the PFI
# method.
# See the full pipeline example under sensitivity_analysis()
# (Requires sensitivity_analysis(methods = "PFI"))
# Final call signature:
# plot_pfi(wrap_object)
```

`plot_pr_curve` *Plotting Precision-Recall Curve*

Description

The **plot_pr_curve()** function generates Precision-Recall curve tracing the relationship between precision and recall across all classification probability thresholds. Particularly informative for imbalanced datasets where ROC curves may be misleading, as PR curves remain sensitive to class distribution changes and provide intuitive performance assessment when one class is substantially rarer than the other.

Usage

```
plot_pr_curve(analysis_object)
```

Arguments

`analysis_object`
: Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

`plot_calibration_curve`

Examples

```
# See the full pipeline example under plot_calibration_curve()
# Final call signature:
# plot_pr_curve(wrap_object)
```

plot_residuals_distribution *Plotting Residuals Distribution*

Description

The **plot_residuals_distribution()** function generates histogram and kernel density visualizations of residuals for regression models on training and test datasets. Enables assessment of residual normality through visual inspection of histogram shape, detection of systematic biases indicating omitted variables or model specification errors, and identification of heavy tails suggesting outliers or influential observations.

Usage

```
plot_residuals_distribution(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

table_best_hyperparameters

Examples

```
# Note: For obtaining the residuals distribution plot the user needs to
# complete till fine_tuning( ) function of the MLwrap pipeline.
# See the full pipeline example under table_best_hyperparameters()
# Final call signature:
# plot_residuals_distribution(wrap_object)
```

plot_roc_curve *Plotting ROC Curve*

Description

The **plot_roc_curve()** function plots Receiver Operating Characteristic (ROC) curve displaying true positive rate versus false positive rate across all classification probability thresholds. Computes Area Under Curve (AUC) as an aggregate discrimination performance metric independent of threshold selection, providing comprehensive assessment of classifier discrimination ability across the entire decision boundary range.

Usage

```
plot_roc_curve(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

plot_calibration_curve

Examples

```
# Note: For obtaining roc curve plot the user needs to
# complete till fine_tuning( ) function of the MLwrap pipeline and
# only with categorical outcome.
# See the full pipeline example under plot_calibration_curve()
# Final call signature:
# plot_roc_curve(wrap_object)
```

---

plot_scatter_predictions

Plotting Observed vs Predictions

Description

The **plot_scatter_predictions()** function generates scatter plots with 45-degree reference lines comparing observed values (vertical axis) against model predictions (horizontal axis) for training and test data. Enables visual assessment of prediction accuracy through distance from the reference line, identification of systematic bias patterns, detection of heteroscedastic prediction errors, and quantification of generalization performance gaps between training and test sets.

Usage

```
plot_scatter_predictions(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

table_best_hyperparameters

Examples

```
# Note: For obtaining the observed vs. predicted values plot the user needs
# to complete till fine_tuning( ) function of the MLwrap pipeline.
# See the full pipeline example under table_best_hyperparameters()
# Final call signature:
# plot_scatter_predictions(wrap_object)
```

`plot_scatter_residuals`

Plotting Residuals vs Predictions

Description

The **plot_scatter_residuals()** function Visualizes residuals plotted against fitted values to detect violations of ordinary least squares assumptions including homoscedasticity (constant error variance), linearity, and independence. Identifies heteroscedastic patterns (non-constant variance across the predictor range), systematic curvature indicating omitted polynomial terms, and outlier points with extreme residual magnitudes.

Usage

```
plot_scatter_residuals(analysis_object)
```

Arguments

`analysis_object`
: Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

table_best_hyperparameters

Examples

```
# Note: For obtaining the residuals vs. predicted values plot the user needs
# to complete till fine_tuning( ) function of the MLwrap pipeline.
# See the full pipeline example under table_best_hyperparameters()
# Final call signature:
# plot_scatter_residuals(wrap_object)
```

plot_shap *Plotting SHAP Plots*

Description

The **plot_shap()** function implements comprehensive SHAP (SHapley Additive exPlanations) value visualizations where SHAP values represent each feature's marginal contribution to model output based on cooperative game theory principles. Provides four visualization modalities: bar plots of mean absolute SHAP values ranking features by average impact magnitude, directional plots showing feature-value correlation with SHAP magnitude and sign, box plots illustrating SHAP value distributions across instances, and swarm plots combining individual prediction contributions with distributional information.

Usage

```
plot_shap(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "SHAP")'.
show_table	Boolean. Whether to print SHAP summarized results table.

Value

analysis_object

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the SHAP plots the user needs to complete till
# sensitivity_analysis( ) function of the MLwrap pipeline using the SHAP
# method.
# See the full pipeline example under sensitivity_analysis()
# (Requires sensitivity_analysis(methods = "SHAP"))
# Final call signature:
# plot_shap(wrap_object)
```

plot_sobol_jansen *Plotting Sobol-Jansen Values Barplot*

Description

The **plot_sobol_jansen()** function displays first-order and total-order Sobol indices decomposing total output variance into contributions from individual features and higher-order interaction terms. Implements variance-based global sensitivity analysis providing comprehensive understanding of feature contributions to output uncertainty, with application restricted to continuous predictor variables.

Usage

```
plot_sobol_jansen(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "Sobol_Jansen")'.
show_table	Boolean. Whether to print Sobol-Jansen results table.

Value

analysis_object

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the Sobol_Jansen plot the user needs to complete till
# sensitivity_analysis( ) function of the MLwrap pipeline using
# the Sobol_Jansen method.
# See the full pipeline example under sensitivity_analysis()
# (Requires sensitivity_analysis(methods = "Sobol_Jansen"))
# Final call signature:
# plot_sobol_jansen(wrap_object)
```

plot_tuning_results *Plotting Tuner Search Results*

Description

The **plot_tuning_results()** function Visualizes hyperparameter optimization search results adapting output format to the optimization methodology employed. For Bayesian Optimization: displays iteration-by-iteration loss function evolution across iterations, acquisition function values guiding sequential hyperparameter sampling, and final hyperparameter configuration with cross-validation performance metrics. For Grid Search: displays performance surfaces across hyperparameter dimensions and rank-ordered configurations by validation performance.

Usage

```
plot_tuning_results(analysis_object)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.

Value

analysis_object

See Also

table_best_hyperparameters

Examples

```
# Note: For obtaining the plot with tuning results the user needs to
# complete till fine_tuning( ) function of the MLwrap pipeline.
# See the full pipeline example under table_best_hyperparameters()
# Final call signature:
# plot_tuning_results(wrap_object)
```

preprocessing *Preprocessing Data Matrix*

Description

The **preprocessing()** function streamlines data preparation for regression and classification tasks by integrating variable selection, type conversion, normalization, and categorical encoding into a single workflow. It takes a data frame and a formula, applies user-specified transformations to numeric and categorical variables using the recipes package, and ensures the outcome variable is properly formatted. The function returns an AnalysisObject containing both the processed data and the transformation pipeline, supporting reproducible and efficient modeling (Kuhn & Wickham, 2020).

Usage

```
preprocessing(
  df,
  formula,
  task = "regression",
  num_vars = NULL,
  cat_vars = NULL,
  norm_num_vars = "all",
  encode_cat_vars = "all",
  y_levels = NULL
)
```

Arguments

df	Input DataFrame. Either a data.frame or tibble.
formula	Modelling Formula. A string of characters or formula.
task	Modelling Task. Either "regression" or "classification".
num_vars	Optional vector of names of the numerical features.
cat_vars	Optional vector of names of the categorical features.
norm_num_vars	Normalize numeric features as z-scores. Either vector of names of numerical features to be normalized or "all" (default).
encode_cat_vars	One Hot Encode Categorical Features. Either vector of names of categorical features to be encoded or "all" (default).
y_levels	Optional ordered vector with names of the target variable levels (Classification task only).

Value

The object returned by the preprocessing function encapsulates a dataset specifically prepared for ML analysis. This object contains the preprocessed data—where variables have been selected, standardized, encoded, and formatted according to the requirements of the chosen modeling task (regression or classification) —as well as a recipes::recipe object that documents all preprocessing steps applied. By automating essential transformations such as normalization, one-hot encoding of categorical variables, and the handling of missing values, the function ensures the data is optimally structured for input into machine learning algorithms. This comprehensive preprocessing not only exposes the underlying structure of the data and reduces the risk of errors, but also provides a robust

foundation for subsequent modeling, validation, and interpretation within the machine learning workflow (Kuhn & Johnson, 2019).

References

Kuhn, M., & Johnson, K. (2019). *Feature Engineering and Selection: A Practical Approach for Predictive Models*. Chapman and Hall/CRC. doi:10.1201/9781315108230

Kuhn, M., & Wickham, H. (2020). *Tidymodels: a collection of packages for modeling and machine learning using tidyverse principles*. https://www.tidymodels.org.

Examples

```
# Example 1: Dataset with preformatted categorical variables
# In this case, internal options for variable types are not needed since
# categorical features are already formatted as factors.

library(MLwrap)

data(sim_data) # sim_data is a simulated dataset with psychological variables

wrap_object <- preprocessing(
          df = sim_data,
         formula = psych_well ~ depression + emot_intel + resilience + life_sat + gender,
          task = "regression"
         )

# Example 2: Dataset where neither the outcome nor the categorical features
# are formatted as factors and all categorical variables are specified to be
# formatted as factors

wrap_object <- preprocessing(
           df = sim_data,
           formula = psych_well_bin ~ gender + depression + age + life_sat,
           task = "classification",
           cat_vars = c("gender")
         )
```

`sensitivity_analysis` *Perform Sensitivity Analysis and Interpretable ML methods*

Description

As the final step in the MLwrap package workflow, this function performs Sensitivity Analysis (SA) on a fitted ML model stored in an `analysis_object` (in the examples, e.g., tidy_object). It evaluates the importance of features using various methods such as Permutation Feature Importance (PFI), SHAP (SHapley Additive exPlanations), Integrated Gradients, Olden sensitivity analysis, and Sobol indices. The function generates numerical results and visualizations (e.g., bar plots, box plots, beeswarm plots) to help interpret the impact of each feature on the model's predictions for both regression and classification tasks, providing critical insights after model training and evaluation.

Following the steps of data preprocessing, model fitting, and performance assessment in the MLwrap pipeline, *sensitivity_analysis()* processes the training and test data using the preprocessing recipe stored in the analysis_object, applies the specified SA methods, and stores the results within the analysis_object. It supports different metrics for evaluation and handles multi-class classification by producing class-specific analyses and plots, ensuring a comprehensive understanding of model behavior (Iooss & Lemaître, 2015).

Usage

```
sensitivity_analysis(
  analysis_object,
  methods = c("PFI"),
  metric = NULL,
  use_test = FALSE
)
```

Arguments

`analysis_object`	analysis_object created from fine_tuning function.
`methods`	Method to be used. A string of the method name: "PFI" (Permutation Feature Importance), "SHAP" (SHapley Additive exPlanations), "Integrated Gradients" (Neural Network only), "Olden" (Neural Networks only), "Sobol_Jansen" (only when all input features are continuous), "Friedman H-stat" (Friedman's H-statistics for feature interaction).
`metric`	Metric used for "PFI" method (Permutation Feature Importance). A string of the name of metric (see Metrics).
`use_test`	Logical. Compute methods using the test set instead of the training set (default = FALSE).

Details

As the concluding phase of the MLwrap workflow—after data preparation, model training, and evaluation—this function interprets models by quantifying and visualizing feature importance. It validates input with `check_args_sensitivity_analysis()`, preprocesses data using the recipe stored in `analysis_object$transformer`, then calculates feature importance via the specified `methods`:

- **PFI (Permutation Feature Importance):** Assesses importance by shuffling feature values and measuring the change in model performance (using the specified or default `metric`).
- **SHAP (SHapley Additive exPlanations):** Computes SHAP values to explain individual predictions by attributing contributions to each feature.
- **Integrated Gradients:** Evaluates feature importance by integrating gradients of the model's output with respect to input features.
- **Olden:** Calculates sensitivity based on connection weights, typically for neural network models, to determine feature contributions.

- **Sobol_Jansen:** Variance-based global sensitivity analysis that decomposes model output variance into contributions from individual features and their interactions. Quantifies how much each feature accounts for prediction variability. Only for continuous outcomes. Estimates first-order and total-order Sobol indices using the Jansen (1999) Monte Carlo estimator.
- **Friedman H-stat:** Computes the Friedman H-statistic for **each feature**. It measures the strength of interaction effects relative to main effects, following the formulation in *Interpretable Machine Learning* (Christoph Molnar). After ranking features by global H-statistic, the top 5 features are selected and **all their pairwise interactions** are computed, returning both **raw interaction strength** and **normalized interaction scores** (0–1).

For classification tasks with more than two outcome levels, the function generates separate results and plots for each class. Visualizations include bar plots for importance metrics, box plots for distribution of values, and beeswarm plots for detailed feature impact across observations. All results are stored in the `analysis_object` under the `sensitivity_analysis` slot, finalizing the MLwrap pipeline with a deep understanding of model drivers.

Value

An updated `analysis_object` containing sensitivity analysis results. Results are stored in the `sensitivity_analysis` slot as a list, with each method's results accessible by name. Generates bar, box, and beeswarm plots for feature importance visualization, completing the workflow with actionable insights.

References

Iooss, B., & Lemaître, P. (2015). A review on global sensitivity analysis methods. In: G. Dellino & C. Meloni (Eds.), *Uncertainty Management in Simulation-Optimization of Complex Systems. Operations Research/Computer Science Interfaces Series* (vol. 59). Springer, Boston, MA. doi:10.1007/9781489975478_5

Jansen, M. J. W. (1999). Analysis of variance designs for model output. *Computer Physics Communications, 117*(1-2), 35–43. doi:10.1016/S00104655(98)001544

Molnar, C. (2022). *Interpretable Machine Learning*. https://christophm.github.io/interpretable-ml-book/

Examples

```
# Example: Using PFI

wrap_object <- preprocessing(
       df = sim_data,
       formula = psych_well ~ depression + life_sat,
       task = "regression"
       )
wrap_object <- build_model(
               analysis_object = wrap_object,
               model_name = "Random Forest",
               hyperparameters = list(
                                 mtry = 2,
                                 trees = 3
                                 )
```

```
                            )
set.seed(123) # For reproducibility
wrap_object <- fine_tuning(wrap_object,
                tuner = "Grid Search CV",
                metrics = c("rmse")
                )
wrap_object <- sensitivity_analysis(wrap_object, methods = "PFI")

# Extracting Results

table_pfi <- table_pfi_results(wrap_object)
```

`sim_data` *sim_data*

Description

This dataset, included in the MLwrap package, is a simulated dataset (Martínez-García et al., 2025) designed to capture relationships among psychological and demographic variables influencing psychological wellbeing, the primary outcome variable. It comprises data for 1,000 individuals.

Usage

```
data(sim_data)
```

Format

A data frame with 1,000 rows and 10 columns:

psych_well Psychological Wellbeing Indicator. Continuous with (0,100)

psych_well_bin Psychological Wellbeing Binary Indicator. Factor with ("Low", "High")

psych_well_pol Psychological Wellbeing Polytomic Indicator. Factor with ("Low", "Somewhat", "Quite a bit", "Very Much")

gender Patient Gender. Factor ("Female", "Male")

age Patient Age. Continuous (18, 85)

socioec_status Socioeconomial Status Indicator. Factor ("Low", "Medium", "High")

emot_intel Emotional Intelligence Indicator. Continuous (24, 120)

resilience Resilience Indicator. Continuous (4, 20)

depression Depression Indicator. Continuous (0, 63)

life_sat Life Satisfaction Indicator. Continuous (5, 35)

Details

The predictor variables include gender (50.7% female), age (range: 18-85 years, mean = 51.63, median = 52, SD = 17.11), and socioeconomic status, categorized as Low (n = 343), Medium (n = 347), and High (n = 310). Additional predictors (features) are emotional intelligence (range: 24-120, mean = 71.97, median = 71, SD = 23.79), resilience (range: 4-20, mean = 11.93, median = 12, SD = 4.46), life satisfaction (range: 5-35, mean = 20.09, median = 20, SD = 7.42), and depression (range: 0-63, mean = 31.45, median = 32, SD = 14.85). The primary outcome variable is emotional wellbeing, measured on a scale from 0 to 100 (mean = 50.22, median = 49, SD = 24.45).

The dataset incorporates correlations as conditions for the simulation. Psychological wellbeing is positively correlated with emotional intelligence (r = 0.50), resilience (r = 0.40), and life satisfaction (r = 0.60), indicating that higher levels of these factors are associated with better emotional health outcomes. Conversely, a strong negative correlation exists between depression and psychological wellbeing (r = -0.80), suggesting that higher depression scores are linked to lower emotional wellbeing. Age shows a slight positive correlation with emotional wellbeing (r = 0.15), reflecting the expectation that older individuals might experience greater emotional stability. Gender and socioeconomic status are included as potential predictors, but the simulation assumes no statistically significant differences in psychological wellbeing across these categories.

Additionally, the dataset includes categorical transformations of psychological wellbeing into binary and polytomous formats: a binary version ("Low" = 477, "High" = 523) and a polytomous version with four levels: "Low" (n = 161), "Somewhat" (n = 351), "Quite a bit" (n = 330), and "Very much" (n = 158). The polytomous transformation uses the 25th, 50th, and 75th percentiles as thresholds for categorizing psychological wellbeing scores. These transformations enable analyses using machine learning models for regression (continuous outcome) and classification (binary or polytomous outcomes) tasks.

Test Performance Exceeding Training Performance

If machine learning models, including SVMs, show better evaluation metrics on the test set than the training set, this anomaly usually signals methodological issues rather than genuine model quality. Typical causes reported in the literature (Hastie et al., 2017) include:

- **Statistical variance in small samples**: Random train-test splits may produce partitions where the test set contains easier-to-classify examples by chance, especially with small sample sizes or difficult tasks (Vabalas et al., 2019; An et al., 2021).
- **Synthetic data characteristics**: Simulated data may contain artificial patterns or non-uniform distributions that create easier test sets compared to training sets.
- **Excessive regularization**: High regularization parameters may limit model capacity to fit training data while paradoxically generalizing better to simpler test patterns, indicating underfitting.
- **Train-test contamination**: Preprocessing (scaling, normalization) performed before train-test split leaks statistical information from test to train, producing overoptimistic performance estimates (Kapoor & Narayanan, 2023).
- **Kernel-data interaction**: Inappropriate kernel parameters may create decision boundaries that better fit test distribution than training distribution.

MLwrap implementation: MLwrap's hyperparameter optimization (via Bayesian Optimization or Grid Search CV) implements 5-fold cross-validation during the tuning process, which provides

more robust parameter selection than single train-test splits. Users should examine evaluation metrics across both training and test sets, and review diagnostic plots (residuals, predictions) to identify potential distribution differences between partitions. When working with small datasets where partition variability may be substantial, running the complete workflow with different random seeds can help assess the stability of results and conclusions. The `sim_data` dataset included in MLwrap is a simulated matrix provided for demonstration purposes only. As synthetic data, it may occasionally exhibit some of these anomalous phenomena (e.g., better test than training performance) due to artificial patterns in the data generation process. Users working with real-world data should always verify results through careful examination of evaluation metrics and diagnostic plots across multiple runs.

References

An, C., Park, Y. W., Ahn, S. S., Han, K., Kim, H., & Lee, S. K. (2021). Radiomics machine learning study with a small sample size: Single random training-test set split may lead to unreliable results. *PLOS ONE*, *16*(8), e0256152. doi:10.1371/journal.pone.0256152

Hastie, T., Tibshirani, R., & Friedman, J. (2017). *The elements of statistical learning: Data mining, inference, and prediction* (2nd ed., corrected 12th printing, Chapter 7). Springer. doi:10.1007/9780387848587

Kapoor, S., & Narayanan, A. (2023). Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, *4*(9), 100804. doi:10.1016/j.patter.2023.100804

Martínez-García, J., Montaño, J. J., Jiménez, R., Gervilla, E., Cajal, B., Núñez, A., Leguizamo, F., & Sesé, A. (2025). Decoding Artificial Intelligence: A Tutorial on Neural Networks in Behavioral Research. *Clinical and Health, 36*(2), 77-95. doi:10.5093/clh2025a13

Vabalas, A., Gowen, E., Poliakoff, E., & Casson, A. J. (2019). Machine learning algorithm validation with a limited sample size. *PLOS ONE*, *14*(11), e0224365. doi:10.1371/journal.pone.0224365

`table_best_hyperparameters`

Best Hyperparameters Configuration

Description

The **table_best_hyperparameters()** function extracts and presents the optimal hyperparameter configuration identified during the model fine-tuning process. This function validates that the model has been properly trained and that hyperparameter tuning has been performed, combining both constant and optimized hyperparameters to generate a comprehensive table with the configuration that maximizes performance according to the specified primary metric. The function includes optional interactive visualization capabilities through the show_table parameter.

Usage

```
table_best_hyperparameters(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.
show_table	Boolean. Whether to print the table.

Value

Tibble with best hyperparameter configuration.

Examples

```
# Note: For obtaining hyoperparameters table the user needs to
# complete till fine_tuning( ) function.

set.seed(123) # For reproducibility
wrap_object <- preprocessing(df = sim_data[1:300 ,],
                             formula = psych_well ~ depression + resilience,
                             task = "regression")
wrap_object <- build_model(wrap_object, "Random Forest",
                           hyperparameters = list(mtry = 2, trees = 3))
wrap_object <- fine_tuning(wrap_object, "Grid Search CV")

# And then, you can obtain the best hyperparameters table.

table_best_hyp <- table_best_hyperparameters(wrap_object)
```

---

table_evaluation_results

Evaluation Results

Description

The **table_evaluation_results()** function provides access to trained model evaluation metrics, automatically adapting to the type of problem being analyzed. For binary classification problems, it returns a unified table with performance metrics, while for multiclass classification it generates separate tables for training and test data, enabling comparative performance evaluation and detection of potential overfitting.

Usage

```
table_evaluation_results(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'fine_tuning()'.
show_table	Boolean. Whether to print the table.

Value

Tibble or list of tibbles (multiclass classification) with evaluation results.

See Also

table_best_hyperparameters

Examples

```
# Note: For obtaining the evaluation table the user needs to
# complete till fine_tuning( ) function.
# See the full pipeline example under table_best_hyperparameters()
# Final call signature:
# table_evaluation_results(wrap_object)
```

---

table_h2_total *Friedman's H-Statistic Table*

Description

The **table_h2_total()** function computes the **global Friedman H-statistic** for each feature, quantifying how much of a variable's predictive contribution arises from interactions with other features rather than from its individual main effect. This metric provides a model-agnostic measure of overall interaction strength, following the formulation presented in *Interpretable Machine Learning* by Christoph Molnar.

The resulting table ranks all features by their global H-statistic, helping identify which predictors participate most in interaction-driven behavior.

Usage

```
table_h2_total(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	A fitted wrap_object with results from sensitivity_analysis(methods = "Friedman H-stat") or compatible internal computations.
show_table	Logical. If TRUE, prints the table (default = FALSE).

Value

A dataframe containing the global H-statistic for each feature.

References

Molnar, C. (2022). *Interpretable Machine Learning*.
https://christophm.github.io/interpretable-ml-book/

See Also

sensitivity_analysis

Examples

```
# After running sensitivity_analysis(wrap_object, methods = "Friedman H-stat"):
# table_h2 <- table_h2_total(wrap_object)
```

`table_integrated_gradients_results`

Integrated Gradients Summarized Results Table

Description

The **table_integrated_gradients_results()** function implements a summarized metrics scheme for Integrated Gradients values. This methodology, specifically designed for neural networks, calculates feature importance through gradient integration along paths from baseline to input. Three different metrics are computed:

- **Mean Absolute Value**
- **Standard Deviation of Mean Absolute Value**
- **Directional Sensitivity Value** (Cov(Feature values, IG values) / Var(Feature values))

Usage

```
table_integrated_gradients_results(analysis_object, show_table = FALSE)
```

Arguments

`analysis_object`	Fitted analysis_object with 'sensitivity_analysis(methods = "Integrated Gradients")'.
`show_table`	Boolean. Whether to print the table.

Value

Tibble or list of tibbles (multiclass classification) with Integrated Gradient summarized results.

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the table with Integrated Gradients method results
# the user needs to complete till sensitivity_analysis() function of the
# MLwrap pipeline using the Integrated Gradient method.
# See the full pipeline example under sensitivity_analysis
# (Requires sensitivity_analysis(methods = "Integrated Gradients"))
# Final call signature:
# table_integrated_gradients_results(wrap_object)
```

---

table_olden_results *Olden Results Table*

Description

The **table_olden_results()** function extracts results from the Olden method, a technique specific to neural networks that calculates relative importance of input variables through analysis of connection weights between network layers. This method provides a measure of each variable's contribution based on the magnitude and direction of synaptic connections.

Usage

```
table_olden_results(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "Olden")'.
show_table	Boolean. Whether to print the table.

Value

Tibble or list of tibbles (multiclass classification) with Olden results.

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the table with Olden method results the user needs to
# complete till sensitivity_analysis() function of the MLwrap pipeline using
# the Olden method. Remember Olden method only can be used with neural
# network model.
# See the full pipeline example under sensitivity_analysis
# (Requires sensitivity_analysis(methods = "Olden"))
# Final call signature:
# table_olden_results(wrap_object)
```

table_pairwise_interaction *Friedman's H-Statistic Pairwise Interaction Table*

Description

The **table_pairwise_interaction()** function computes **pairwise interaction strengths** between predictors using **Friedman's H-statistic**, following the formulation described in *Interpretable Machine Learning* by Christoph Molnar. While the global H-statistic summarizes the overall interaction strength of each individual feature, this function focuses specifically on **pairwise feature interactions**, quantifying how strongly two variables interact in influencing the model's predictions. If `normalize = TRUE`, interaction scores are returned on a **0–1 scale** for ease of comparison; if `FALSE`, raw interaction magnitudes are returned.

Usage

```
table_pairwise_interaction(
  analysis_object,
  show_table = FALSE,
  normalize = TRUE
)
```

Arguments

`analysis_object`	A fitted `wrap_object` with results from `sensitivity_analysis(methods = "Friedman H-stat")` with pairwise interactions computed internally.
`show_table`	Logical. If TRUE, prints the resulting interaction table to the console (default = FALSE).
`normalize`	Logical. If TRUE (default), returns **normalized** pairwise interaction strengths; if FALSE, returns **raw** interaction values.

Value

A tibble containing pairwise interaction strengths between all feature pairs, in either normalized or raw form depending on the `normalize` argument.

References

Molnar, C. (2022). *Interpretable Machine Learning*.
https://christophm.github.io/interpretable-ml-book/

See Also

`sensitivity_analysis`, `table_h2_total`

Examples

```
# After running:
# wrap_object <- sensitivity_analysis(wrap_object, methods = "Friedman H-stat")
#
# Obtain normalized pairwise interactions:
# table_norm <- table_pairwise_interaction(wrap_object)
#
# Obtain raw interaction strengths:
# table_raw <- table_pairwise_interaction(wrap_object, normalize = FALSE)
```

---

table_pfi_results *Permutation Feature Importance Results Table*

Description

The **table_pfi_results()** function extracts Permutation Feature Importance results, a model-agnostic technique that evaluates variable importance through performance degradation when randomly permuting each feature's values.

Usage

```
table_pfi_results(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "PFI")'.
show_table	Boolean. Whether to print the table.

Value

Tibble or list of tibbles (multiclass classification) with PFI results.

Examples

```
# Note: For obtaining the table with PFI method results the user needs to
# complete till sensitivity_analysis() function of the
# MLwrap pipeline using PFI method

set.seed(123) # For reproducibility
wrap_object <- preprocessing(df = sim_data[1:300 ,],
                             formula = psych_well ~ depression + emot_intel,
                             task = "regression")
wrap_object <- build_model(wrap_object, "Random Forest",
                           hyperparameters = list(mtry = 2, trees = 3))
wrap_object <- fine_tuning(wrap_object, "Grid Search CV")
wrap_object <- sensitivity_analysis(wrap_object, methods = "PFI")
```

```
# And then, you can obtain the PFI results table.

table_pfi <- table_pfi_results(wrap_object)
```

`table_shap_results` *SHAP Summarized Results Table*

Description

The **table_shap_results()** function processes previously calculated SHAP (SHapley Additive exPlanations) values and generates summarized metrics including mean absolute value, standard deviation of mean absolute value, and a directional sensitivity value calculated as the covariance between feature values and SHAP values divided by the variance of feature values. This directional metric provides information about the nature of the relationship between each variable and model predictions. To summarize the SHAP values calculated, three different metrics are computed:

- **Mean Absolute Value**
- **Standard Deviation of Mean Absolute Value**
- **Directional Sensitivity Value** (Cov(Feature values, SHAP values) / Var(Feature values))

Usage

```
table_shap_results(analysis_object, show_table = FALSE)
```

Arguments

`analysis_object`	Fitted analysis_object with 'sensitivity_analysis(methods = "SHAP")'.
`show_table`	Boolean. Whether to print the table.

Value

Tibble or list of tibbles (multiclass classification) with SHAP summarized results.

See Also

`sensitivity_analysis`

Examples

```
# Note: For obtaining the table with SHAP method results the user needs
# to complete till sensitivity_analysis() function of the
# MLwrap pipeline using the SHAP method.
# See the full pipeline example under sensitivity_analysis
# (Requires sensitivity_analysis(methods = "SHAP"))
# Final call signature:
# table_shap_results(wrap_object)
```

table_sobol_jansen_results *Sobol-Jansen Results Table*

Description

The **table_sobol_jansen_results()** function processes results from Sobol-Jansen global sensitivity analysis, a variance decomposition-based methodology that quantifies each variable's contribution and their interactions to the total variability of model predictions. This technique is particularly valuable for identifying higher-order effects and complex interactions between variables.

Usage

```
table_sobol_jansen_results(analysis_object, show_table = FALSE)
```

Arguments

analysis_object	Fitted analysis_object with 'sensitivity_analysis(methods = "Sobol_Jansen")'.
show_table	Boolean. Whether to print the table.

Value

Tibble or list of tibbles (multiclass classification) with Sobol-Jansen results.

See Also

sensitivity_analysis

Examples

```
# Note: For obtaining the table with Sobol_Jansen method results the user
# needs to complete till sensitivity_analysis() function of the MLwrap
# pipeline using the Sobol_Jansen method. Sobol_Jansen method only works
# when all input features are continuous.
# See the full pipeline example under sensitivity_analysis
# (Requires sensitivity_analysis(methods = "Sobol_Jansen"))
# Final call signature:
# table_sobol_jansen_results(wrap_object)
```

`tutorial` *MLwrap Comprehensive Tutorial*

Description

A comprehensive tutorial demonstrating the complete MLwrap workflow is available. The tutorial provides detailed guidance on data preprocessing, model building, hyperparameter tuning, model evaluation, and sensitivity analysis across all supported machine learning algorithms (Neural Networks, Random Forests, SVM, and XGBoost) within the Knowledge Discovery in Databases (KDD) framework.

Usage

```
MLwrap_tutorial()
```

Details

Citation: Jiménez, R., Martínez-García, J., Montaño, J. J., & Sesé, A. (2025). *MLwrap: Simplifying Machine Learning workflows in R. PsyarXiv.* doi:10.31234/osf.io/j6m4z_v2

Value

Character string with the arXiv URL

Preprint

Available at doi:10.31234/osf.io/j6m4z_v2

Why consult the tutorial

While MLwrap provides a streamlined and user-friendly interface for implementing machine learning workflows, the underlying models represent sophisticated algorithms with substantial theoretical and computational complexity. The tutorial bridges this gap by explaining the rationale behind preprocessing decisions, hyperparameter choices, and interpretation of model outputs. Understanding these concepts ensures appropriate application of the methods, proper interpretation of results, and awareness of potential limitations in specific contexts.

The tutorial demonstrates practical applications through complete workflows, helping users navigate the balance between methodological rigor and implementation simplicity that MLwrap offers. This is particularly valuable for researchers transitioning from traditional statistical methods to machine learning approaches, or those seeking to ensure reproducible and theoretically sound applications in their work.

Users are strongly encouraged to consult the tutorial for detailed examples and best practices.

Tutorial for implementing ML with Python

This paper is also interesting for ML users as it serves as a primer for estimating ML models using Python code, particularly in the context of Social, Health, and Behavioral research.

Martínez-García, J., Montaño, J. J., Jiménez, R., Gervilla, E., Cajal, B., Núñez, A., Leguizamo, F., & Sesé, A. (2025). Decoding Artificial Intelligence: A Tutorial on Neural Networks in Behavioral Research. *Clinical and Health, 36*(2), 77-95. doi:10.5093/clh2025a13

Examples

```
MLwrap_tutorial()
```

Index