Package 'MLwrap'

October 21, 2025

Title Machine Learning Modelling for Everyone

Version 0.2.1

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(). 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, vip, 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 2025-10-21 15:10:02 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

set.seed(123) # For reproducibility
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

set.seed(123) # For reproducibility
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 different tuning strategies, such as **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, verbose = FALSE)
```

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).
`verbose`	Whether to show tuning process. Boolean TRUE or FALSE (default).

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, Singapore. https://doi.org/10.1007/978-981-19-5170-1

Examples

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

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

`plot_calibration_curve` *Plotting Calibration Curve*

Description

The **plot_calibration_curve()** function is specifically designed for binary classification and produces calibration curves that evaluate correspondence between predicted probabilities and observed frequencies. This function is restricted to binary classification problems and provides crucial information about the reliability of the model's probabilistic estimates.

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.

set.seed(123) # For reproducibility
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))
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 for both training and test data in classification problems. This visualization allows evaluation of classification accuracy by category and identification of confusion patterns between classes, providing insights into which classes are most frequently misclassified.

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 generates distributions of model output scores segmented by class, facilitating evaluation of separability between categories and identification of problematic overlaps. This visualization helps assess whether the model produces sufficiently distinct score distributions for different classes.

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()** function implements specialized visualizations for evaluating model effectiveness in marketing and case selection contexts. The gain curve shows cumulative gains as a function of population percentile, helping assess how well the model identifies high-value cases in ranked populations.

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

Plots a graph visualization of the Neural Network's architecture along with its optimized hyperparameters.

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 replicates the SHAP visualization structure for integrated gradient values, providing the same four graphical modalities adapted to this specific interpretability methodology for neural networks. This function is particularly valuable for understanding feature importance in deep learning architectures where gradients provide direct information about model sensitivity.

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 produces lift curves that display the lift factor as a function of population percentile. This visualization is particularly useful for direct marketing applications, showing how much better the model performs compared to random selection at different population segments.

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

Plots the training loss curve of the Neural Network model on the validation set. This plot can be used for underfitting / overfitting diagnostics.

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 generates specialized bar plots for visualizing Olden method results, which provide importance measures specific to neural networks based on connection weight analysis. This method offers insights into how input variables influence predictions through the network's synaptic connections.

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_pfi *Plotting Permutation Feature Importance Barplot*

Description

The **plot_pfi()** function generates bar plots to visualize feature importance through permutation, providing clear representation of each predictor variable's relative contribution to model performance. The function includes an option to display accompanying numerical results tables for comprehensive interpretation.

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 curves, which are particularly valuable for evaluating classifier performance on imbalanced datasets. These curves show the relationship between precision and recall across different decision thresholds, complementing ROC curve analysis.

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 histograms of residual distributions for both training and test data in regression problems. This visualization enables evaluation of error normality and detection of systematic patterns in model residuals. The function uses patchwork to combine training and test plots in a single display for direct comparison.

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 produces ROC (Receiver Operating Characteristic) curves, providing fundamental visual metrics for evaluating binary and multiclass classifier performance. The ROC curve illustrates the trade-off between true positive rate and false positive rate across different classification thresholds.

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 between observed and predicted values, providing direct visual assessment of model predictive accuracy. The function displays both training and test results side by side, enabling evaluation of model generalization performance and identification of potential overfitting.

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 produces scatter plots relating residuals to predictions, facilitating identification of heteroscedasticity and non-linear patterns in model errors. This diagnostic plot is essential for validating regression model assumptions and detecting potential issues with model specification or data quality.

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 a comprehensive set of visualizations for SHAP values, including bar plots of mean absolute values, directional plots showing positive or negative contribution nature, box plots illustrating SHAP value distributions by variable, and swarm plots combining individual and distributional information. This multifaceted approach enables deep understanding of how each feature influences model predictions.

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 produces bar plots for Sobol-Jansen analysis results, offering a global sensitivity perspective based on variance decomposition. This methodology is particularly valuable for identifying higher-order effects and complex interactions between variables in model predictions.

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 generates graphical representations of hyperparameter search results, automatically adapting to the type of optimizer used. When Bayesian optimization is employed, the function presents additional plots showing the iterative evolution of the loss function and search results throughout the optimization process. This function validates that model fitting has been completed and that hyperparameter tuning was actually performed before attempting to display results.

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 (1st ed.)*. Chapman and Hall/CRC. https://doi.org/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)
```

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 Network only), "Sobol_Jansen" (only when all input features are continuous).
`metric`	Metric used for "PFI" method (Permutation Feature Importance). A string of the name of metric (see Metrics).

Details

As the concluding phase of the MLwrap workflow—after data preparation, model training, and evaluation—this function enables users to interpret their models by quantifying and visualizing feature importance. It first validates the input arguments using `check_args_sensitivity_analysis()`. Then, it preprocesses the training and test data using the recipe stored in `analysis_object$transformer`. Depending on the specified `methods`, it calculates feature importance using:

- **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:** Performs variance-based global sensitivity analysis by decomposing the model output variance into contributions from individual features and their interactions, quantifying how much each feature and combination of features accounts for the variability in predictions. Only for continuous outcomes, not for categorical. Specifically, estimates first-order and total-order Sobol' sensitivity indices simultaneously using the Jansen (1999) Monte Carlo estimator.

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` with the results of the sensitivity analysis stored in the `sensitivity_analysis` slot as a list. Each method's results are accessible under named elements (e.g., `sensitivity_analysis[["PFI"]]`). Additionally, the function produces various plots (bar plots, box plots, beeswarm plots) for visual interpretation of feature importance, tailored to the task type and number of outcome levels, completing the MLwrap workflow with actionable model insights.

References

Iooss, B., & Lemaître, P. (2015). A review on global sensitivity analysis methods. In C. Meloni & G. Dellino (Eds.), *Uncertainty Management in Simulation-Optimization of Complex Systems: Algorithms and Applications* (pp. 101-122). Springer. https://doi.org/10.1007/978-1-4899-7547-8_5

Jansen, M. J. W. (1999). Analysis of variance designs for model output. Computer Physics Communications, 117(1-2), 35–43. https://doi.org/10.1016/S0010-4655(98)00154-4

Examples

```
# Example: Using PFI

set.seed(123) # For reproducibility
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
```

```
                                  )
                              )
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 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 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.

Note

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.

References

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

`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_integrated_gradients_results

Integrated Gradients Summarized Results Table

Description

The **table_integrated_gradients_results()** function implements the same summarized metrics scheme for Integrated Gradients values, a methodology specifically designed for neural networks that calculates feature importance through gradient integration along paths from a baseline to the current input. To summarize the Integrated Gradients values calculated, 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_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)
```

Index