Package ‘E2E’

August 26, 2025

Title Ensemble Learning Framework for Diagnostic and Prognostic Modeling

Version 0.0.3

Description Provides a framework to build and evaluate diagnosis or prognosis models using stacking, voting, and bagging ensemble techniques with various base learners. The package also includes tools for visualization and interpretation of models. The development version of the package is available on 'GitHub' at <https://github.com/xiaojie0519/E2E>. The methods are based on the foundational work of Breiman (1996) <doi:10.1007/BF00058655> on bagging and Wolpert (1992) <doi:10.1016/S0893-6080(05)80023-1> on stacking.

License MIT + file LICENSE

Encoding UTF-8

URL https://xiaojie0519.github.io/E2E/

BugReports https://github.com/xiaojie0519/E2E/issues

RoxygenNote 7.3.2

Imports caret, dplyr, gbm, ggplot2, glmnet, magrittr, MASS, patchwork, pROC, PRROC, randomForestSRC, readr, RSNNS, shapviz, survcomp, survival, survivalROC, survminer, timeROC, xgboost

Suggests ada, doParallel, e1071, kernlab, klaR, knitr, nnet, randomForest, RColorBrewer, rmarkdown, rpart

Depends R (>= 3.5)

LazyData true

VignetteBuilder knitr

Language en

NeedsCompilation no

Author Shanjie Luan [aut, cre]

Maintainer Shanjie Luan <Luan20050519@163.com>

Repository CRAN

Date/Publication 2025-08-26 19:30:14 UTC

Contents

`apply_dia` *Apply a Trained Diagnostic Model to New Data*

Description

Applies a previously trained model (or ensemble) to a new, unseen dataset to generate predicted probabilities.

Usage

```
apply_dia(
  trained_model_object,
  new_data,
  label_col_name = NULL,
  pos_class,
  neg_class
)
```

Arguments

`trained_model_object`
A trained model object, as returned by `models_dia`, `bagging_dia`, `stacking_dia`, `voting_dia`, or `imbalance_dia`.

`new_data` A data frame containing the new data for prediction. The first column must be the sample ID, subsequent columns are features.

`label_col_name` A character string, the name of the column containing the class labels in the new data. This is optional and only used to include true labels in the output; it is not used for prediction.

`pos_class` A character string, the label for the positive class (must match the label used during training).

`neg_class` A character string, the label for the negative class (must match the label used during training).

Value

A data frame with `sample` (ID), `label` (original numeric label from new data, or NA if not provided), and `score` (predicted probability for the positive class).

Examples

```
# 1. Assume 'train_dia' and 'test_dia' are loaded from your package
# data(train_dia)
# data(test_dia) # test_dia has same structure, maybe without the label column
initialize_modeling_system_dia()

# 2. Train a model
train_results <- models_dia(
  data = train_dia, model = "lasso",
  new_positive_label = "Case", new_negative_label = "Control"
)
trained_lasso_model <- train_results$lasso$model_object

# 3. Apply the trained model to new data
new_predictions <- apply_dia(
  trained_model_object = trained_lasso_model,
  new_data = test_dia,
  label_col_name = "Disease_Status", # Optional
  pos_class = "Case",
  neg_class = "Control"
)
utils::head(new_predictions)
```

---

apply_pro — *Apply a Trained Prognostic Model to New Data*

Description

Applies a previously trained prognostic model (or ensemble) to a new, unseen dataset to generate prognostic scores.

Usage

```
apply_pro(trained_model_object, new_data, time_unit = "day")
```

Arguments

`trained_model_object`
: A trained model object, as returned by `models_pro`, `bagging_pro`, or `stacking_pro`.

`new_data`
: A data frame containing the new data for prediction. It should follow the same structure as the training data: ID, Outcome, Time, Features. The outcome and time columns are used for data preparation and can be included in the output, but the model's prediction only uses the features. If outcome/time are unknown, they can be filled with NA.

`time_unit`
: A character string, the unit of time in the third column of `new_data`.

Value

A data frame with ID, outcome, time, and predicted score for the new data.

See Also

evaluate_model_pro

Examples

```
# NOTE: This example requires 'train_pro' and 'test_pro' datasets.
if (requireNamespace("E2E", quietly = TRUE) &&
    "train_pro" %in% utils::data(package = "E2E")$results[,3] &&
    "test_pro" %in% utils::data(package = "E2E")$results[,3]) {

  data(train_pro, package = "E2E")
  data(test_pro, package = "E2E")
  initialize_modeling_system_pro()

  train_results <- models_pro(data = train_pro, model = "lasso_pro")
  trained_lasso_model <- train_results$lasso_pro$model_object

  # Apply the trained model to new data
  new_data_predictions <- apply_pro(
    trained_model_object = trained_lasso_model,
    new_data = test_pro,
    time_unit = "day" # Specify time unit of test_pro
  )
  utils::head(new_data_predictions)
}
```

---

bagging_dia   *Train a Bagging Diagnostic Model*

Description

Implements a Bagging (Bootstrap Aggregating) ensemble for diagnostic models. It trains multiple base models on bootstrapped samples of the training data and aggregates their predictions by averaging probabilities.

Usage

```
bagging_dia(
  data,
  base_model_name,
  n_estimators = 50,
  subset_fraction = 0.632,
  tune_base_model = FALSE,
  threshold_strategy = "default",
```

```
  specific_threshold_value = 0.5,
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Positive",
  new_negative_label = "Negative",
  seed = 456
)
```

Arguments

- `data` — A data frame where the first column is the sample ID, the second is the outcome label, and subsequent columns are features.
- `base_model_name` — A character string, the name of the base diagnostic model to use (e.g., "rf", "lasso"). This model must be registered.
- `n_estimators` — An integer, the number of base models to train.
- `subset_fraction` — A numeric value between 0 and 1, the fraction of samples to bootstrap for each base model.
- `tune_base_model` — Logical, whether to enable tuning for each base model.
- `threshold_strategy` — A character string (e.g., "f1", "youden", "default") or a numeric value (0-1) for determining the evaluation threshold for the ensemble.
- `specific_threshold_value` — A numeric value between 0 and 1. Only used if `threshold_strategy` is "numeric".
- `positive_label_value` — A numeric or character value in the raw data representing the positive class.
- `negative_label_value` — A numeric or character value in the raw data representing the negative class.
- `new_positive_label` — A character string, the desired factor level name for the positive class (e.g., "Positive").
- `new_negative_label` — A character string, the desired factor level name for the negative class (e.g., "Negative").
- `seed` — An integer, for reproducibility.

Value

A list containing the `model_object`, `sample_score`, and `evaluation_metrics`.

See Also

`initialize_modeling_system_dia`, `evaluate_model_dia`

Examples

```
# This example assumes your package includes a dataset named 'train_dia'.
# If not, create a toy data frame first.
if (exists("train_dia")) {
  initialize_modeling_system_dia()

  bagging_rf_results <- bagging_dia(
    data = train_dia,
    base_model_name = "rf",
    n_estimators = 5, # Reduced for a quick example
    threshold_strategy = "youden",
    positive_label_value = 1,
    negative_label_value = 0,
    new_positive_label = "Case",
    new_negative_label = "Control"
  )
  print_model_summary_dia("Bagging (RF)", bagging_rf_results)
}
```

---

bagging_pro *Train a Bagging Prognostic Model*

Description

Implements a Bagging (Bootstrap Aggregating) ensemble for prognostic models. It trains multiple base models on bootstrapped samples of the training data and aggregates their predictions.

Usage

```
bagging_pro(
  data,
  base_model_name,
  n_estimators = 10,
  subset_fraction = 0.632,
  tune_base_model = FALSE,
  time_unit = "day",
  years_to_evaluate = c(1, 3, 5),
  seed = 456
)
```

Arguments

`data` A data frame for training. The first column must be the sample ID, the second column the event status (0/1), the third column the time, and subsequent columns the features.

`base_model_name` A character string, the name of the base prognostic model to use (e.g., "lasso_pro", "rsf_pro"). This model must be registered.

n_estimators	An integer, the number of base models to train.
subset_fraction	A numeric value between 0 and 1, the fraction of samples to bootstrap for each base model.
tune_base_model	Logical, whether to enable tuning for each base model.
time_unit	A character string, the unit of time in the third column of data.
years_to_evaluate	A numeric vector of specific years at which to calculate time-dependent AUROC for evaluation.
seed	An integer, for reproducibility.

Value

A list containing the model_object, sample_score, and evaluation_metrics.

See Also

initialize_modeling_system_pro, evaluate_model_pro

Examples

```
# NOTE: This example requires the 'train_pro' dataset.
if (requireNamespace("E2E", quietly = TRUE) &&
"train_pro" %in% utils::data(package = "E2E")$results[,3]) {
  data(train_pro, package = "E2E")
  initialize_modeling_system_pro()

  bagging_lasso_results <- bagging_pro(
    data = train_pro,
    base_model_name = "lasso_pro",
    n_estimators = 3, # Small number for example speed
    subset_fraction = 0.8,
    years_to_evaluate = c(1, 3)
  )
  print_model_summary_pro("Bagging (Lasso)", bagging_lasso_results)
}
```

---

calculate_metrics_at_threshold_dia

Calculate Classification Metrics at a Specific Threshold

Description

Calculates various classification performance metrics (Accuracy, Precision, Recall, F1-score, Specificity, True Positives, etc.) for binary classification at a given probability threshold.

Usage

```
calculate_metrics_at_threshold_dia(
  prob_positive,
  y_true,
  threshold,
  pos_class,
  neg_class
)
```

Arguments

`prob_positive`	A numeric vector of predicted probabilities for the positive class.
`y_true`	A factor vector of true class labels.
`threshold`	A numeric value between 0 and 1, the probability threshold above which a prediction is considered positive.
`pos_class`	A character string, the label for the positive class.
`neg_class`	A character string, the label for the negative class.

Value

A list containing:

- `Threshold`: The threshold used.
- `Accuracy`: Overall prediction accuracy.
- `Precision`: Precision for the positive class.
- `Recall`: Recall (Sensitivity) for the positive class.
- `F1`: F1-score for the positive class.
- `Specificity`: Specificity for the negative class.
- `TP`, `TN`, `FP`, `FN`, `N`: Counts of True Positives, True Negatives, False Positives, False Negatives, and total samples.

Examples

```
y_true_ex <- factor(c("Negative", "Positive", "Positive", "Negative", "Positive"),
                    levels = c("Negative", "Positive"))
prob_ex <- c(0.1, 0.8, 0.6, 0.3, 0.9)
metrics <- calculate_metrics_at_threshold_dia(
  prob_positive = prob_ex,
  y_true = y_true_ex,
  threshold = 0.5,
  pos_class = "Positive",
  neg_class = "Negative"
)
print(metrics)
```

dt_dia *Train a Decision Tree Model for Classification*

Description

Trains a single Decision Tree model using `caret::train` (via `rpart` method) for binary classification.

Usage

```
dt_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning for `cp` (complexity parameter) (if `TRUE`) or use a fixed value (if `FALSE`).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained Decision Tree model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
dt_model <- dt_dia(X_toy, y_toy)
print(dt_model)
```

en_dia — *Train an Elastic Net (L1 and L2 Regularized Logistic Regression) Model for Classification*

Description

Trains an Elastic Net-regularized logistic regression model using `caret::train` (via `glmnet` method) for binary classification.

Usage

```
en_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning for `lambda` (if `TRUE`) or use a fixed value (if `FALSE`). `alpha` is fixed at 0.5 for Elastic Net.
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained Elastic Net model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
en_model <- en_dia(X_toy, y_toy)
print(en_model)
```

en_pro *Train an Elastic Net Cox Proportional Hazards Model*

Description

Trains a Cox proportional hazards model with Elastic Net regularization using glmnet (with alpha = 0.5).

Usage

```
en_pro(X, y_surv, tune = FALSE)
```

Arguments

X	A data frame of features.
y_surv	A survival::Surv object representing the survival outcome.
tune	Logical, whether to perform hyperparameter tuning (currently simplified/ignored for direct cv.glmnet usage which inherently tunes lambda).

Value

A list of class "train" containing the trained glmnet model object, names of features used in training, and model type. The returned object also includes fitted_scores (linear predictor), y_surv, best_lambda, and alpha_val.

Examples

```
set.seed(42)
n_samples <- 50
n_features <- 10
X_data <- as.data.frame(matrix(rnorm(n_samples * n_features), ncol = n_features))
Y_surv_obj <- survival::Surv(
  time = runif(n_samples, 100, 1000),
  event = sample(0:1, n_samples, replace = TRUE)
)

# Train the model
en_model <- en_pro(X_data, Y_surv_obj)
print(en_model$finalModel)
```

evaluate_model_dia *Evaluate Diagnostic Model Performance*

Description

Evaluates the performance of a trained diagnostic model using various metrics relevant to binary classification, including AUROC, AUPRC, and metrics at an optimal or specified probability threshold.

Usage

```
evaluate_model_dia(
  model_obj = NULL,
  X_data = NULL,
  y_data,
  sample_ids,
  threshold_strategy = c("default", "f1", "youden", "numeric"),
  specific_threshold_value = 0.5,
  pos_class,
  neg_class,
  precomputed_prob = NULL,
  y_original_numeric = NULL
)
```

Arguments

`model_obj` A trained model object (typically a `caret::train` object or a list from an ensemble like Bagging). Can be `NULL` if `precomputed_prob` is provided.

`X_data` A data frame of features corresponding to the data used for evaluation. Required if `model_obj` is provided and `precomputed_prob` is `NULL`.

`y_data` A factor vector of true class labels for the evaluation data.

`sample_ids` A vector of sample IDs for the evaluation data.

`threshold_strategy` A character string, defining how to determine the threshold for class-specific metrics: "default" (0.5), "f1" (maximizes F1-score), "youden" (maximizes Youden's J statistic), or "numeric" (uses `specific_threshold_value`).

`specific_threshold_value` A numeric value between 0 and 1. Only used if `threshold_strategy` is "numeric".

`pos_class` A character string, the label for the positive class.

`neg_class` A character string, the label for the negative class.

`precomputed_prob` Optional. A numeric vector of precomputed probabilities for the positive class. If provided, `model_obj` and `X_data` are not used for score derivation.

y_original_numeric
Optional. The original numeric/character vector of labels. If not provided, it's inferred from y_data using global pos_label_value and neg_label_value.

Value

A list containing:

- sample_score: A data frame with sample (ID), label (original numeric), and score (predicted probability for positive class).
- evaluation_metrics: A list of performance metrics:
  - Threshold_Strategy: The strategy used for threshold selection.
  - _Threshold: The chosen probability threshold.
  - Accuracy, Precision, Recall, F1, Specificity: Metrics calculated at _Threshold.
  - AUROC: Area Under the Receiver Operating Characteristic curve.
  - AUROC_95CI_Lower, AUROC_95CI_Upper: 95% confidence interval for AUROC.
  - AUPRC: Area Under the Precision-Recall curve.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))
ids_toy <- paste0("Sample", 1:n_obs)

# 2. Train a model
rf_model <- rf_dia(X_toy, y_toy)

# 3. Evaluate the model using F1-score optimal threshold
eval_results <- evaluate_model_dia(
  model_obj = rf_model,
  X_data = X_toy,
  y_data = y_toy,
  sample_ids = ids_toy,
  threshold_strategy = "f1",
  pos_class = "Case",
  neg_class = "Control"
)
str(eval_results)
```

evaluate_model_pro *Evaluate Prognostic Model Performance*

Description

Evaluates the performance of a trained prognostic model using various metrics relevant to survival analysis, including C-index, time-dependent AUROC, and Kaplan-Meier (KM) group analysis (Hazard Ratio and p-value).

Usage

```
evaluate_model_pro(
  trained_model_obj = NULL,
  X_data = NULL,
  Y_surv_obj,
  sample_ids,
  years_to_evaluate = c(1, 3, 5),
  precomputed_score = NULL,
  meta_normalize_params = NULL
)
```

Arguments

`trained_model_obj`
: A trained model object (of class "train" as returned by model training functions like `lasso_pro`, `rsf_pro`, etc.). Can be `NULL` if `precomputed_score` is provided.

`X_data`
: A data frame of features corresponding to the data used for evaluation. Required if `trained_model_obj` is provided and `precomputed_score` is `NULL`.

`Y_surv_obj`
: A `survival::Surv` object for the evaluation data.

`sample_ids`
: A vector of sample IDs for the evaluation data.

`years_to_evaluate`
: A numeric vector of specific years at which to calculate time-dependent AUROC.

`precomputed_score`
: Optional. A numeric vector of precomputed prognostic scores for the samples. If provided, `trained_model_obj` and `X_data` are not strictly necessary for score derivation.

`meta_normalize_params`
: Optional. A list of normalization parameters (min/max values) used for base model scores in a stacking ensemble. Used when `trained_model_obj` is a stacking model to ensure consistent normalization during prediction.

Value

A list containing:

- sample_score: A data frame with ID, outcome, time, and score columns.
- evaluation_metrics: A list of performance metrics:
  - C_index: Harrell's C-index.
  - AUROC_Years: A named list of time-dependent AUROC values for specified years.
  - AUROC_Average: The average of time-dependent AUROC values.
  - KM_HR: Hazard Ratio for High vs Low risk groups (split by median score).
  - KM_P_value: P-value for the KM group comparison.
  - KM_Cutoff: The score cutoff used to define High/Low risk groups (median score).

Examples

```
# Generate dummy data
set.seed(42)
n <- 50
X <- as.data.frame(matrix(rnorm(n * 5), n, 5))
Y_surv <- survival::Surv(time = runif(n, 1, 5*365), event = sample(0:1, n, TRUE))
ids <- paste0("s", 1:n)

# Train a simple model
initialize_modeling_system_pro()
model_obj <- lasso_pro(X, Y_surv)

# Evaluate the model on the same data
eval_results <- evaluate_model_pro(
  trained_model_obj = model_obj,
  X_data = X,
  Y_surv_obj = Y_surv,
  sample_ids = ids,
  years_to_evaluate = c(1, 2, 3)
)
str(eval_results$evaluation_metrics)
```

---

evaluate_predictions_pro *Evaluate Prognostic Predictions*

Description

A convenience wrapper to evaluate a data frame of prognostic predictions. This function is ideal for evaluating the output of apply_pro.

Usage

```
evaluate_predictions_pro(prediction_df, years_to_evaluate = c(1, 3, 5))
```

Arguments

prediction_df
: A data frame containing predictions. Must include columns named `ID`, `outcome`, `time`, and `score`. This format matches the output of `apply_pro`.

years_to_evaluate
: A numeric vector of specific years at which to calculate time-dependent AUROC.

Value

A list of evaluation metrics, including C-index, time-dependent AUROC, and Kaplan-Meier analysis results.

See Also

`apply_pro`, `evaluate_model_pro`

Examples

```
# Assume 'trained_model' and 'test_pro' data are available
if (requireNamespace("E2E", quietly = TRUE) &&
    "train_pro" %in% utils::data(package = "E2E")$results[,3] &&
    "test_pro" %in% utils::data(package = "E2E")$results[,3]) {

  data(train_pro, package = "E2E")
  data(test_pro, package = "E2E")
  initialize_modeling_system_pro()
  model_results <- models_pro(data = train_pro, model = "lasso_pro")

  # 1. Get predictions on new data
  predictions <- apply_pro(model_results$lasso_pro$model_object, test_pro)

  # 2. Evaluate these predictions using the simplified function
 evaluation_metrics <- evaluate_predictions_pro(predictions, years_to_evaluate = c(1, 3))
  print(evaluation_metrics)
}
```

`figure_dia` *Plot Diagnostic Model Evaluation Figures*

Description

Generates and returns a ggplot object for Receiver Operating Characteristic (ROC) curves, Precision-Recall (PRC) curves, or confusion matrices.

Usage

```
figure_dia(type, data, file = NULL)
```

Arguments

type
: String, specifies the type of plot to generate. Options are "roc", "prc", or "matrix".

data
: A list object containing model evaluation results. It must include:
  - `sample_score`: A data frame with "label" (0/1) and "score" columns.
  - `evaluation_metrics`: A list with a "Final_Threshold" or "Final_Threshold" value.

file
: Optional. A string specifying the path to save the plot (e.g., "plot.png"). If `NULL` (the default), the plot object is returned instead of being saved.

Value

A ggplot object. If the `file` argument is provided, the plot is also saved to the specified path.

Examples

```
# Create example data for a diagnostic model
external_eval_example_dia <- list(
  sample_score = data.frame(
    ID = paste0("S", 1:100),
    label = sample(c(0, 1), 100, replace = TRUE),
    score = runif(100, 0, 1)
  ),
  evaluation_metrics = list(
    Final_Threshold = 0.53
  )
)

# Generate an ROC curve plot object
roc_plot <- figure_dia(type = "roc", data = external_eval_example_dia)
# To display the plot, simply run:
# print(roc_plot)

# Generate a PRC curve and save it to a temporary file
# tempfile() creates a safe, temporary path as required by CRAN
temp_prc_path <- tempfile(fileext = ".png")
figure_dia(type = "prc", data = external_eval_example_dia, file = temp_prc_path)

# Generate a Confusion Matrix plot
matrix_plot <- figure_dia(type = "matrix", data = external_eval_example_dia)
```

---

figure_pro *Plot Prognostic Model Evaluation Figures*

Description

Generates and returns a ggplot object for Kaplan-Meier (KM) survival curves or time-dependent ROC curves.

Usage

```
figure_pro(type, data, file = NULL, time_unit = "days")
```

Arguments

`type`	"km" or "tdroc"
`data`	list with: • sample_score: data.frame(time, outcome, score) • evaluation_metrics: for "km" needs KM_Cutoff; for "tdroc" needs AUROC_Years (numeric years like c(1,3,5), OR a named vector/list like c('1'=0.74,'3'=0.82,'5'=0.85))
`file`	optional path to save
`time_unit`	"days" (default), "months", or "years" for df$time

Value

ggplot object

`figure_shap` *Generate and Plot SHAP Explanation Figures*

Description

Creates SHAP (SHapley Additive exPlanations) plots to explain feature contributions by training a surrogate model on the original model's scores.

Usage

```
figure_shap(data, raw_data, target_type, file = NULL, model_type = "xgboost")
```

Arguments

`data`	A list containing `sample_score`, a data frame with sample IDs and `score`.
`raw_data`	A data frame with original features. The first column must be the sample ID.
`target_type`	String, the analysis type: "diagnosis" or "prognosis". This determines which columns in `raw_data` are treated as features.
`file`	Optional. A string specifying the path to save the plot. If `NULL` (default), the plot object is returned.
`model_type`	String, the surrogate model for SHAP calculation. "xgboost" (default) or "lasso".

Value

A patchwork object combining SHAP summary and importance plots. If `file` is provided, the plot is also saved.

Examples

```
# --- Example for a Diagnosis Model ---
set.seed(123)
train_dia_data <- data.frame(
  SampleID = paste0("S", 1:100),
  Label = sample(c(0, 1), 100, replace = TRUE),
  FeatureA = rnorm(100, 10, 2),
  FeatureB = runif(100, 0, 5)
)
model_results <- list(
  sample_score = data.frame(ID = paste0("S", 1:100), score = runif(100, 0, 1))
)

# Generate SHAP plot object
shap_plot <- figure_shap(
  data = model_results,
  raw_data = train_dia_data,
  target_type = "diagnosis",
  model_type = "xgboost"
)
# To display the plot:
# print(shap_plot)
```

`find_optimal_threshold_dia`

Find Optimal Probability Threshold

Description

Determines an optimal probability threshold for binary classification based on maximizing F1-score or Youden's J statistic.

Usage

```
find_optimal_threshold_dia(
  prob_positive,
  y_true,
  type = c("f1", "youden"),
  pos_class,
  neg_class
)
```

Arguments

`prob_positive`	A numeric vector of predicted probabilities for the positive class.
`y_true`	A factor vector of true class labels.

type	A character string, specifying the optimization criterion: "f1" for F1-score or "youden" for Youden's J statistic (Sensitivity + Specificity - 1).
pos_class	A character string, the label for the positive class.
neg_class	A character string, the label for the negative class.

Value

A numeric value, the optimal probability threshold.

Examples

```
y_true_ex <- factor(c("Negative", "Positive", "Positive", "Negative", "Positive"),
                    levels = c("Negative", "Positive"))
prob_ex <- c(0.1, 0.8, 0.6, 0.3, 0.9)

# Find threshold maximizing F1-score
opt_f1_threshold <- find_optimal_threshold_dia(
  prob_positive = prob_ex,
  y_true = y_true_ex,
  type = "f1",
  pos_class = "Positive",
  neg_class = "Negative"
)
print(opt_f1_threshold)

# Find threshold maximizing Youden's J
opt_youden_threshold <- find_optimal_threshold_dia(
  prob_positive = prob_ex,
  y_true = y_true_ex,
  type = "youden",
  pos_class = "Positive",
  neg_class = "Negative"
)
print(opt_youden_threshold)
```

---

gbm_dia *Train a Gradient Boosting Machine (GBM) Model for Classification*

Description

Trains a Gradient Boosting Machine (GBM) model using `caret::train` for binary classification.

Usage

```
gbm_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

X	A data frame of features.
y	A factor vector of class labels.
tune	Logical, whether to perform hyperparameter tuning for interaction.depth, n.trees, and shrinkage (if TRUE) or use fixed values (if FALSE).
cv_folds	An integer, the number of cross-validation folds for caret.

Value

A caret::train object representing the trained GBM model.

Examples

```
set.seed(42)
n_obs <- 200
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
gbm_model <- gbm_dia(X_toy, y_toy)
print(gbm_model)
```

---

gbm_pro — *Train a Gradient Boosting Machine (GBM) for Survival Data*

Description

Trains a Gradient Boosting Machine (GBM) model with a Cox proportional hazards loss function using gbm.

Usage

```
gbm_pro(X, y_surv, tune = FALSE, cv.folds = 3)
```

Arguments

X	A data frame of features.
y_surv	A survival::Surv object representing the survival outcome.
tune	Logical, whether to perform simplified hyperparameter tuning. If TRUE, n.trees, interaction.depth, and shrinkage are set to predefined values suitable for tuning; otherwise, default values are used.
cv.folds	Integer. The number of cross-validation folds to use. Setting this to 0 or 1 will disable cross-validation. Defaults to 3.

Value

A list of class "train" containing the trained gbm model object, names of features used in training, and model type. The returned object also includes `fitted_scores` (linear predictor), `y_surv`, and `best_iter`.

Examples

```
# Generate some dummy survival data
set.seed(42)
n_samples <- 200
n_features <- 5
X_data <- as.data.frame(matrix(rnorm(n_samples * n_features), ncol = n_features))
Y_surv_obj <- survival::Surv(
  time = runif(n_samples, 100, 1000),
  event = sample(0:1, n_samples, replace = TRUE)
)

# Train the model for the example *without* cross-validation to pass R CMD check
# In real use, you might use the default cv.folds = 3
gbm_model <- gbm_pro(X_data, Y_surv_obj, cv.folds = 0)
print(gbm_model$finalModel)
```

---

get_registered_models_dia

Get Registered Diagnostic Models

Description

Retrieves a list of all diagnostic model functions currently registered in the internal environment.

Usage

```
get_registered_models_dia()
```

Value

A named list where names are the registered model names and values are the corresponding model functions.

See Also

`register_model_dia`, `initialize_modeling_system_dia`

Examples

```
# Ensure system is initialized to see the default models
initialize_modeling_system_dia()
models <- get_registered_models_dia()
# See available model names
print(names(models))
```

---

get_registered_models_pro

Get Registered Prognostic Models

Description

Retrieves a list of all prognostic model functions currently registered in the internal environment.

Usage

```
get_registered_models_pro()
```

Value

A named list where names are the registered model names and values are the corresponding model functions.

See Also

register_model_pro, initialize_modeling_system_pro

Examples

```
# Get all currently registered models
initialize_modeling_system_pro() # Ensure system is initialized
models <- get_registered_models_pro()
names(models) # See available model names
```

---

imbalance_dia

Train an EasyEnsemble Model for Imbalanced Classification

Description

Implements the EasyEnsemble algorithm. It trains multiple base models on balanced subsets of the data (by undersampling the majority class) and aggregates their predictions.

Usage

```
imbalance_dia(
  data,
  base_model_name = "xb",
  n_estimators = 10,
  tune_base_model = FALSE,
  threshold_choices = "default",
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Positive",
  new_negative_label = "Negative",
  seed = 456
)
```

Arguments

`data` A data frame where the first column is the sample ID, the second is the outcome label, and subsequent columns are features.

`base_model_name` A character string, the name of the base diagnostic model to use (e.g., "xb", "rf"). This model must be registered.

`n_estimators` An integer, the number of base models to train (number of subsets).

`tune_base_model` Logical, whether to enable tuning for each base model.

`threshold_choices` A character string (e.g., "f1", "youden", "default") or a numeric value (0-1) for determining the evaluation threshold for the ensemble.

`positive_label_value` A numeric or character value in the raw data representing the positive class.

`negative_label_value` A numeric or character value in the raw data representing the negative class.

`new_positive_label` A character string, the desired factor level name for the positive class (e.g., "Positive").

`new_negative_label` A character string, the desired factor level name for the negative class (e.g., "Negative").

`seed` An integer, for reproducibility.

Value

A list containing the `model_object`, `sample_score`, and `evaluation_metrics`.

See Also

`initialize_modeling_system_dia`, `evaluate_model_dia`

Examples

```
# 1. Initialize the modeling system
initialize_modeling_system_dia()

# 2. Create an imbalanced toy dataset
set.seed(42)
n_obs <- 100
n_minority <- 10
data_imbalanced_toy <- data.frame(
  ID = paste0("Sample", 1:n_obs),
  Status = c(rep(1, n_minority), rep(0, n_obs - n_minority)),
  Feat1 = rnorm(n_obs),
  Feat2 = runif(n_obs)
)

# 3. Run the EasyEnsemble algorithm
# n_estimators is reduced for a quick example
easyensemble_results <- imbalance_dia(
  data = data_imbalanced_toy,
  base_model_name = "xb",
  n_estimators = 3,
  threshold_choices = "f1"
)
print_model_summary_dia("EasyEnsemble (XGBoost)", easyensemble_results)
```

---

initialize_modeling_system_dia *Initialize Diagnostic Modeling System*

Description

Initializes the diagnostic modeling system by loading required packages and registering default diagnostic models (Random Forest, XGBoost, SVM, MLP, Lasso, Elastic Net, Ridge, LDA, QDA, Naive Bayes, Decision Tree, GBM). This function should be called once before using `models_dia()` or ensemble methods.

Usage

```
initialize_modeling_system_dia()
```

Value

Invisible NULL. Initializes the internal model registry.

Examples

```
# Initialize the system (typically run once at the start of a session or script)
initialize_modeling_system_dia()

# Check if a default model like Random Forest is now registered
"rf" %in% names(get_registered_models_dia())
```

`initialize_modeling_system_pro` *Initialize Prognostic Modeling System*

Description

Initializes the prognostic modeling system by loading required packages and registering default prognostic models (Lasso, Elastic Net, Ridge, Random Survival Forest, Stepwise Cox, GBM for Cox). This function should be called once before using `run_models_pro()` or ensemble methods.

Usage

```
initialize_modeling_system_pro()
```

Value

Invisible NULL. Initializes the internal model registry.

Examples

```
# Initialize the system (typically run once at the start of a session or script)
initialize_modeling_system_pro()

# Check if models are now registered
print(names(get_registered_models_pro()))
```

`lasso_dia` *Train a Lasso (L1 Regularized Logistic Regression) Model for Classification*

Description

Trains a Lasso-regularized logistic regression model using `caret::train` (via `glmnet` method) for binary classification.

Usage

```
lasso_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning for `lambda` (if `TRUE`) or use a fixed value (if `FALSE`). `alpha` is fixed at 1 for Lasso.
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained Lasso model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
lasso_model <- lasso_dia(X_toy, y_toy)
print(lasso_model)
```

`lasso_pro` *Train a Lasso Cox Proportional Hazards Model*

Description

Trains a Cox proportional hazards model with Lasso regularization using `glmnet`.

Usage

```
lasso_pro(X, y_surv, tune = FALSE)
```

Arguments

`X`	A data frame of features.
`y_surv`	A `survival::Surv` object representing the survival outcome.
`tune`	Logical, whether to perform hyperparameter tuning (currently simplified/ignored for direct `cv.glmnet` usage which inherently tunes lambda).

Value

A list of class "train" containing the trained `glmnet` model object, names of features used in training, and model type. The returned object also includes `fitted_scores` (linear predictor) and `y_surv`.

Examples

```
set.seed(42)
n_samples <- 50
n_features <- 10
X_data <- as.data.frame(matrix(rnorm(n_samples * n_features), ncol = n_features))
Y_surv_obj <- survival::Surv(
  time = runif(n_samples, 100, 1000),
  event = sample(0:1, n_samples, replace = TRUE)
)

# Train the model
lasso_model <- lasso_pro(X_data, Y_surv_obj)
print(lasso_model$finalModel)
```

`lda_dia` *Train a Linear Discriminant Analysis (LDA) Model for Classification*

Description

Trains a Linear Discriminant Analysis (LDA) model using `caret::train` for binary classification.

Usage

```
lda_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning (currently ignored for LDA).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained LDA model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
lda_model <- lda_dia(X_toy, y_toy)
print(lda_model)
```

`load_and_prepare_data_dia`
Load and Prepare Data for Diagnostic Models

Description

Loads a CSV file containing patient data, extracts features, and converts the label column into a factor suitable for classification models. Handles basic data cleaning like trimming whitespace and type conversion.

Usage

```
load_and_prepare_data_dia(
  data_path,
  label_col_name,
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Positive",
  new_negative_label = "Negative"
)
```

Arguments

`data_path` A character string, the file path to the input CSV data. The first column is assumed to be a sample ID.

`label_col_name` A character string, the name of the column containing the class labels.

`positive_label_value` A numeric or character value that represents the positive class in the raw data.

`negative_label_value` A numeric or character value that represents the negative class in the raw data.

new_positive_label
: A character string, the desired factor level name for the positive class (e.g., "Positive").

new_negative_label
: A character string, the desired factor level name for the negative class (e.g., "Negative").

Value

A list containing:

- X: A data frame of features (all columns except ID and label).
- y: A factor vector of class labels, with levels new_negative_label and new_positive_label.
- sample_ids: A vector of sample IDs (the first column of the input data).
- pos_class_label: The character string used for the positive class factor level.
- neg_class_label: The character string used for the negative class factor level.
- y_original_numeric: The original numeric/character vector of labels.

Examples

```
# Create a dummy CSV file in a temporary directory for demonstration
temp_csv_path <- tempfile(fileext = ".csv")
dummy_data <- data.frame(
  ID = paste0("Patient", 1:50),
  Disease_Status = sample(c(0, 1), 50, replace = TRUE),
  FeatureA = rnorm(50),
  FeatureB = runif(50, 0, 100),
  CategoricalFeature = sample(c("X", "Y", "Z"), 50, replace = TRUE)
)
write.csv(dummy_data, temp_csv_path, row.names = FALSE)

# Load and prepare data from the temporary file
prepared_data <- load_and_prepare_data_dia(
  data_path = temp_csv_path,
  label_col_name = "Disease_Status",
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Case",
  new_negative_label = "Control"
)

# Check prepared data structure
str(prepared_data$X)
table(prepared_data$y)

# Clean up the dummy file
unlink(temp_csv_path)
```

load_and_prepare_data_pro

Load and Prepare Data for Prognostic Models

Description

Loads a CSV file containing patient data, extracts features, outcome, and time columns, and prepares them into a format suitable for survival analysis models. Handles basic data cleaning like NA removal and column type conversion.

Usage

```
load_and_prepare_data_pro(
  data_path,
  outcome_col_name,
  time_col_name,
  time_unit = c("day", "month", "year")
)
```

Arguments

`data_path`	A character string, the file path to the input CSV data. The first column is assumed to be a sample ID.
`outcome_col_name`	A character string, the name of the column containing event status (0 for censored, 1 for event).
`time_col_name`	A character string, the name of the column containing event or censoring time.
`time_unit`	A character string, the unit of time in `time_col_name`. Can be "day", "month", or "year". Times will be converted to days internally.

Value

A list containing:

- `X`: A data frame of features (all columns except ID, outcome, and time).
- `Y_surv`: A `survival::Surv` object created from time and outcome.
- `sample_ids`: A vector of sample IDs (the first column of the input data).
- `outcome_numeric`: A numeric vector of outcome status.
- `time_numeric`: A numeric vector of time, converted to days.

Examples

```
temp_csv_path <- tempfile(fileext = ".csv")
dummy_data <- data.frame(
  ID = paste0("Patient", 1:50),
  FeatureA = rnorm(50),
  FeatureB = runif(50, 0, 100),
  CategoricalFeature = sample(c("A", "B", "C"), 50, replace = TRUE),
  Outcome_Status = sample(c(0, 1), 50, replace = TRUE),
  Followup_Time_Months = runif(50, 10, 60)
)
write.csv(dummy_data, temp_csv_path, row.names = FALSE)

# Load and prepare data
prepared_data <- load_and_prepare_data_pro(
  data_path = temp_csv_path,
  outcome_col_name = "Outcome_Status",
  time_col_name = "Followup_Time_Months",
  time_unit = "month"
)

# Check prepared data structure
str(prepared_data$X)
print(prepared_data$Y_surv[1:5])

# Clean up dummy file
unlink(temp_csv_path)
```

---

min_max_normalize *Min-Max Normalization*

Description

Normalizes a numeric vector to a range of 0 to 1 using min-max scaling.

Usage

```
min_max_normalize(x, min_val = NULL, max_val = NULL)
```

Arguments

x	A numeric vector to be normalized.
min_val	Optional. The minimum value to use for normalization. If NULL, the minimum of x is used.
max_val	Optional. The maximum value to use for normalization. If NULL, the maximum of x is used.

Value

A numeric vector with values scaled between 0 and 1. If min_val and max_val are equal (or x has no variance), returns a vector of 0.5s.

Examples

```
# Normalize a vector
x_vec <- c(10, 20, 30, 40, 50)
normalized_x <- min_max_normalize(x_vec)
print(normalized_x) # Should be 0, 0.25, 0.5, 0.75, 1

# Normalize with custom min/max
custom_normalized_x <- min_max_normalize(x_vec, min_val = 0, max_val = 100)
print(custom_normalized_x) # Should be 0.1, 0.2, 0.3, 0.4, 0.5
```

---

mlp_dia — *Train a Multi-Layer Perceptron (Neural Network) Model for Classification*

Description

Trains a Multi-Layer Perceptron (MLP) neural network model using `caret::train` for binary classification.

Usage

```
mlp_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning using `caret`'s default grid (if `TRUE`) or a fixed value (if `FALSE`).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained MLP model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
mlp_model <- mlp_dia(X_toy, y_toy)
```

```
print(mlp_model)
```

---

models_dia *Run Multiple Diagnostic Models*

Description

Trains and evaluates one or more registered diagnostic models on a given dataset.

Usage

```
models_dia(
  data,
  model = "all_dia",
  tune = FALSE,
  seed = 123,
  threshold_choices = "default",
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Positive",
  new_negative_label = "Negative"
)
```

Arguments

data	A data frame where the first column is the sample ID, the second is the outcome label, and subsequent columns are features.
model	A character string or vector of character strings, specifying which models to run. Use "all_dia" to run all registered models.
tune	Logical, whether to enable hyperparameter tuning for individual models.
seed	An integer, for reproducibility of random processes.
threshold_choices	A character string (e.g., "f1", "youden", "default") or a numeric value (0-1), or a named list/vector allowing different threshold strategies/values for each model.
positive_label_value	A numeric or character value in the raw data representing the positive class.
negative_label_value	A numeric or character value in the raw data representing the negative class.
new_positive_label	A character string, the desired factor level name for the positive class (e.g., "Positive").
new_negative_label	A character string, the desired factor level name for the negative class (e.g., "Negative").

Value

A named list, where each element corresponds to a run model and contains its trained `model_object`, `sample_score` data frame, and `evaluation_metrics`.

See Also

`initialize_modeling_system_dia`, `evaluate_model_dia`

Examples

```
# This example assumes your package includes a dataset named 'train_dia'.
# If not, you should create a toy data frame similar to the one below.
#
# train_dia <- data.frame(
#   ID = paste0("Patient", 1:100),
#   Disease_Status = sample(c(0, 1), 100, replace = TRUE),
#   FeatureA = rnorm(100),
#   FeatureB = runif(100)
# )

# Ensure the 'train_dia' dataset is available in the environment
# For example, if it is exported by your package:
# data(train_dia)

# Check if 'train_dia' exists, otherwise skip the example
if (exists("train_dia")) {
  # 1. Initialize the modeling system
  initialize_modeling_system_dia()

  # 2. Run selected models
  results <- models_dia(
    data = train_dia,
    model = c("rf", "lasso"), # Run only Random Forest and Lasso
    threshold_choices = list(rf = "f1", lasso = 0.6), # Different thresholds
    positive_label_value = 1,
    negative_label_value = 0,
    new_positive_label = "Case",
    new_negative_label = "Control",
    seed = 42
  )

  # 3. Print summaries
  for (model_name in names(results)) {
    print_model_summary_dia(model_name, results[[model_name]])
  }
}
```

models_pro *Run Multiple Prognostic Models*

Description

Trains and evaluates one or more registered prognostic models on a given dataset.

Usage

```
models_pro(
  data,
  model = "all_pro",
  tune = FALSE,
  seed = 123,
  time_unit = "day",
  years_to_evaluate = c(1, 3, 5)
)
```

Arguments

data	A data frame for training. The first column must be the sample ID, the second column the event status (0/1), the third column the time, and subsequent columns the features.
model	A character string or vector of character strings, specifying which models to run. Use "all_pro" to run all registered models.
tune	Logical, whether to enable hyperparameter tuning for individual models.
seed	An integer, for reproducibility of random processes.
time_unit	A character string, the unit of time in the third column of `data`. Can be "day", "month", or "year".
years_to_evaluate	A numeric vector of specific years at which to calculate time-dependent AUROC.

Value

A named list, where each element corresponds to a run model and contains its trained `model_object`, `sample_score` data frame, and `evaluation_metrics`.

See Also

`initialize_modeling_system_pro`, `evaluate_model_pro`

Examples

```
# NOTE: This example requires the 'train_pro' dataset to be exported by the package.
# If it is not, replace `data(train_pro)` with code to create a dummy dataframe.
# For demonstration, we assume `train_pro` is available.
if (requireNamespace("E2E", quietly = TRUE) &&
 "train_pro" %in% utils::data(package = "E2E")$results[,3]) {
  data(train_pro, package = "E2E")

  # Initialize the modeling system
  initialize_modeling_system_pro()

  # Run selected models
  results <- models_pro(
    data = train_pro,
    model = c("lasso_pro", "rsf_pro"), # Run only Lasso and RSF
    years_to_evaluate = c(1, 3, 5),
    seed = 42
  )

  # Print summaries
  for (model_name in names(results)) {
    print_model_summary_pro(model_name, results[[model_name]])
  }
}
```

`nb_dia` *Train a Naive Bayes Model for Classification*

Description

Trains a Naive Bayes model using `caret::train` for binary classification.

Usage

```
nb_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning using `caret`'s default grid (if `TRUE`) or fixed values (if `FALSE`).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained Naive Bayes model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
nb_model <- nb_dia(X_toy, y_toy)
print(nb_model)
```

---

print_model_summary_dia

Print Diagnostic Model Summary

Description

Prints a formatted summary of the evaluation metrics for a diagnostic model, either from training data or new data evaluation.

Usage

```
print_model_summary_dia(model_name, results_list, on_new_data = FALSE)
```

Arguments

`model_name`	A character string, the name of the model (e.g., "rf", "Bagging (RF)").
`results_list`	A list containing model evaluation results, typically an element from the output of `models_dia()` or the result of `bagging_dia()`, `stacking_dia()`, `voting_dia()`, or `imbalance_dia()`. It must contain `evaluation_metrics` and `model_object` (if applicable).
`on_new_data`	Logical, indicating whether the results are from applying the model to new, unseen data (`TRUE`) or from the training/internal validation data (`FALSE`).

Value

NULL. Prints the summary to the console.

Examples

```
# Example for a successfully evaluated model
successful_results <- list(
  evaluation_metrics = list(
    Threshold_Strategy = "f1",
    `_Threshold` = 0.45,
    AUROC = 0.85, AUROC_95CI_Lower = 0.75, AUROC_95CI_Upper = 0.95,
    AUPRC = 0.80, Accuracy = 0.82, F1 = 0.78,
    Precision = 0.79, Recall = 0.77, Specificity = 0.85
  )
)
print_model_summary_dia("MyAwesomeModel", successful_results)

# Example for a failed model
failed_results <- list(evaluation_metrics = list(error = "Training failed"))
print_model_summary_dia("MyFailedModel", failed_results)
```

`print_model_summary_pro` *Print Prognostic Model Summary*

Description

Prints a formatted summary of the evaluation metrics for a prognostic model, either from training data or new data evaluation.

Usage

```
print_model_summary_pro(model_name, results_list, on_new_data = FALSE)
```

Arguments

`model_name`	A character string, the name of the model (e.g., "lasso_pro").
`results_list`	A list containing model evaluation results, typically an element from the output of `run_models_pro()` or the result of `bagging_pro()`, `stacking_pro()`. It must contain `evaluation_metrics` and `model_object` (if applicable).
`on_new_data`	Logical, indicating whether the results are from applying the model to new, unseen data (`TRUE`) or from the training/internal validation data (`FALSE`).

Value

NULL. Prints the summary to the console.

Examples

```
if (requireNamespace("E2E", quietly = TRUE) &&
 "train_pro" %in% utils::data(package = "E2E")$results[,3]) {
  data(train_pro, package = "E2E")
  initialize_modeling_system_pro()
  results <- models_pro(data = train_pro, model = "lasso_pro")

  # Print summary for the trained model
  print_model_summary_pro("lasso_pro", results$lasso_pro, on_new_data = FALSE)

  # Example for a failed model
  failed_results <- list(evaluation_metrics = list(error = "Training failed"))
  print_model_summary_pro("MyFailedModel", failed_results)
}
```

---

qda_dia — *Train a Quadratic Discriminant Analysis (QDA) Model for Classification*

Description

Trains a Quadratic Discriminant Analysis (QDA) model using `caret::train` for binary classification.

Usage

```
qda_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning (currently ignored for QDA).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained QDA model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
```

```
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
qda_model <- qda_dia(X_toy, y_toy)
print(qda_model)
```

register_model_dia *Register a Diagnostic Model Function*

Description

Registers a user-defined or pre-defined diagnostic model function with the internal model registry. This allows the function to be called later by its registered name, facilitating a modular model management system.

Usage

```
register_model_dia(name, func)
```

Arguments

name	A character string, the unique name to register the model under.
func	A function, the R function implementing the diagnostic model. This function should typically accept `X` (features) and `y` (labels) as its first two arguments and return a `caret::train` object.

Value

NULL. The function registers the model function invisibly.

See Also

get_registered_models_dia, initialize_modeling_system_dia

Examples

```
# Example of a dummy model function for registration
my_dummy_rf_model <- function(X, y, tune = FALSE, cv_folds = 5) {
  message("Training dummy RF model...")
  # This is a placeholder and doesn't train a real model.
  # It returns a list with a structure similar to a caret train object.
  list(method = "dummy_rf")
}

# Initialize the system before registering
initialize_modeling_system_dia()
```

```
# Register the new model
register_model_dia("dummy_rf", my_dummy_rf_model)

# Verify that the model is now in the list of registered models
"dummy_rf" %in% names(get_registered_models_dia())
```

register_model_pro *Register a Prognostic Model Function*

Description

Registers a user-defined or pre-defined prognostic model function with the internal model registry. This allows the function to be called later by its registered name, facilitating a modular model management system.

Usage

```
register_model_pro(name, func)
```

Arguments

name	A character string, the unique name to register the model under.
func	A function, the R function implementing the prognostic model. This function should typically accept X (features) and y_surv (survival object) as its first two arguments.

Value

NULL. The function registers the model function invisibly.

See Also

get_registered_models_pro, initialize_modeling_system_pro

Examples

```
# Example of a dummy model function for registration
my_dummy_cox_model <- function(X, y_surv, tune = FALSE) {
  # A minimal survival model for demonstration
  model_fit <- survival::coxph(y_surv ~ ., data = X)
  structure(list(finalModel = model_fit, X_train_cols = colnames(X),
                 model_type = "survival_dummy_cox"), class = "train")
}

# Register the dummy model
initialize_modeling_system_pro() # Ensure system is initialized
register_model_pro("dummy_cox", my_dummy_cox_model)
"dummy_cox" %in% names(get_registered_models_pro()) # Check if registered
```

`rf_dia` *Train a Random Forest Model for Classification*

Description

Trains a Random Forest model using `caret::train` for binary classification.

Usage

```
rf_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning using `caret`'s default grid (if `TRUE`) or use a fixed `mtry` value (if `FALSE`).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained Random Forest model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
rf_model <- rf_dia(X_toy, y_toy)
print(rf_model)
```

`ridge_dia` *Train a Ridge (L2 Regularized Logistic Regression) Model for Classification*

Description

Trains a Ridge-regularized logistic regression model using `caret::train` (via `glmnet` method) for binary classification.

Usage

```
ridge_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning for `lambda` (if `TRUE`) or use a fixed value (if `FALSE`). `alpha` is fixed at 0 for Ridge.
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained Ridge model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
ridge_model <- ridge_dia(X_toy, y_toy)
print(ridge_model)
```

ridge_pro *Train a Ridge Cox Proportional Hazards Model*

Description

Trains a Cox proportional hazards model with Ridge regularization using `glmnet`.

Usage

```
ridge_pro(X, y_surv, tune = FALSE)
```

Arguments

X	A data frame of features.
y_surv	A `survival::Surv` object representing the survival outcome.
tune	Logical, whether to perform hyperparameter tuning (currently simplified/ignored for direct `cv.glmnet` usage which inherently tunes lambda).

Value

A list of class "train" containing the trained `glmnet` model object, names of features used in training, and model type. The returned object also includes `fitted_scores` (linear predictor), `y_surv`, and `best_lambda`.

Examples

```
set.seed(42)
n_samples <- 50
n_features <- 10
X_data <- as.data.frame(matrix(rnorm(n_samples * n_features), ncol = n_features))
Y_surv_obj <- survival::Surv(
  time = runif(n_samples, 100, 1000),
  event = sample(0:1, n_samples, replace = TRUE)
)

# Train the model
ridge_model <- ridge_pro(X_data, Y_surv_obj)
print(ridge_model$finalModel)
```

rsf_pro	*Train a Random Survival Forest Model*

Description

Trains a Random Survival Forest (RSF) model using `randomForestSRC`.

Usage

```
rsf_pro(X, y_surv, tune = FALSE)
```

Arguments

`X`	A data frame of features.
`y_surv`	A `survival::Surv` object representing the survival outcome.
`tune`	Logical, whether to perform hyperparameter tuning (a simplified message is currently provided, full tuning with `tune.rfsrc` is recommended for advanced use).

Value

A list of class "train" containing the trained `rfsrc` model object, names of features used in training, and model type. The returned object also includes `fitted_scores` and `y_surv`.

Examples

```
# Generate some dummy survival data
set.seed(42)
n_samples <- 50
n_features <- 5
X_data <- as.data.frame(matrix(rnorm(n_samples * n_features), ncol = n_features))
Y_surv_obj <- survival::Surv(
  time = runif(n_samples, 100, 1000),
  event = sample(0:1, n_samples, replace = TRUE)
)

# Train the model (ntree is small for a quick example)
rsf_model <- rsf_pro(X_data, Y_surv_obj)
print(rsf_model$finalModel)
```

stacking_dia *Train a Stacking Diagnostic Model*

Description

Implements a Stacking ensemble. It trains multiple base models, then uses their predictions as features to train a meta-model.

Usage

```
stacking_dia(
  results_all_models,
  data,
  meta_model_name,
  top = 5,
  tune_meta = FALSE,
  threshold_choices = "f1",
  seed = 789,
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Positive",
  new_negative_label = "Negative"
)
```

Arguments

`results_all_models`
: A list of results from `models_dia()`, containing trained base model objects and their evaluation metrics.

`data`
: A data frame where the first column is the sample ID, the second is the outcome label, and subsequent columns are features. Used for training the meta-model.

`meta_model_name`
: A character string, the name of the meta-model to use (e.g., "lasso", "gbm"). This model must be registered.

`top`
: An integer, the number of top-performing base models (ranked by AUROC) to select for the stacking ensemble.

`tune_meta`
: Logical, whether to enable tuning for the meta-model.

`threshold_choices`
: A character string (e.g., "f1", "youden", "default") or a numeric value (0-1) for determining the evaluation threshold for the ensemble.

`seed`
: An integer, for reproducibility.

`positive_label_value`
: A numeric or character value in the raw data representing the positive class.

`negative_label_value`
: A numeric or character value in the raw data representing the negative class.

new_positive_label
: A character string, the desired factor level name for the positive class (e.g., "Positive").

new_negative_label
: A character string, the desired factor level name for the negative class (e.g., "Negative").

Value

A list containing the model_object, sample_score, and evaluation_metrics.

See Also

models_dia, evaluate_model_dia

Examples

```
# 1. Initialize the modeling system
initialize_modeling_system_dia()

# 2. Create a toy dataset for demonstration
set.seed(42)
data_toy <- data.frame(
  ID = paste0("Sample", 1:60),
  Status = sample(c(0, 1), 60, replace = TRUE),
  Feat1 = rnorm(60),
  Feat2 = runif(60)
)

# 3. Generate mock base model results (as if from models_dia)
# In a real scenario, you would run models_dia() on your full dataset
base_model_results <- models_dia(
  data = data_toy,
  model = c("rf", "lasso"),
  seed = 123
)

# 4. Run the stacking ensemble
stacking_results <- stacking_dia(
  results_all_models = base_model_results,
  data = data_toy,
  meta_model_name = "gbm",
  top = 2,
  threshold_choices = "f1"
)
print_model_summary_dia("Stacking (GBM)", stacking_results)
```

stacking_pro *Train a Stacking Prognostic Model*

Description

Implements a Stacking ensemble for prognostic models. It trains multiple base models and uses their predictions to train a meta-model.

Usage

```
stacking_pro(
  results_all_models,
  data,
  meta_model_name,
  top = 3,
  tune_meta = FALSE,
  time_unit = "day",
  years_to_evaluate = c(1, 3, 5),
  seed = 789
)
```

Arguments

`results_all_models`
: A list of results from `models_pro()`, containing trained base model objects and their evaluation metrics.

`data`
: A data frame for training the meta-model. The first column must be ID, second event status (0/1), third time, and subsequent columns features.

`meta_model_name`
: A character string, the name of the meta-model to use (e.g., "lasso_pro", "gbm_pro"). This model must be registered.

`top`
: An integer, the number of top-performing base models (ranked by C-index) to select for the stacking ensemble.

`tune_meta`
: Logical, whether to enable tuning for the meta-model.

`time_unit`
: A character string, the unit of time in the third column of `data`.

`years_to_evaluate`
: A numeric vector of specific years at which to calculate time-dependent AUROC for evaluation.

`seed`
: An integer, for reproducibility.

Value

A list containing the `model_object`, `sample_score`, and `evaluation_metrics`.

See Also

models_pro, evaluate_model_pro

Examples

```
# NOTE: This example requires the 'train_pro' dataset.
if (requireNamespace("E2E", quietly = TRUE) &&
"train_pro" %in% utils::data(package = "E2E")$results[,3]) {
  data(train_pro, package = "E2E")
  initialize_modeling_system_pro()

  # First, generate results for base models
  base_model_results <- models_pro(data = train_pro, model = c("lasso_pro", "rsf_pro"))

  # Then, create the stacking ensemble
  stacking_lasso_results <- stacking_pro(
    results_all_models = base_model_results,
    data = train_pro,
    meta_model_name = "lasso_pro",
    top = 3,
    years_to_evaluate = c(1, 3)
  )
  print_model_summary_pro("Stacking (Lasso)", stacking_lasso_results)
}
```

`stepcox_pro` *Train a Stepwise Cox Proportional Hazards Model*

Description

Trains a Cox proportional hazards model and performs backward stepwise selection using `MASS::stepAIC` to select important features.

Usage

```
stepcox_pro(X, y_surv, tune = FALSE)
```

Arguments

`X`	A data frame of features.
`y_surv`	A `survival::Surv` object representing the survival outcome.
`tune`	Logical, whether to perform hyperparameter tuning (currently ignored).

Value

A list of class "train" containing the trained `coxph` model object after stepwise selection, names of features used in training, and model type. The returned object also includes `fitted_scores` (linear predictor) and `y_surv`.

Examples

```
set.seed(42)
n_samples <- 50
n_features <- 5
X_data <- as.data.frame(matrix(rnorm(n_samples * n_features), ncol = n_features))
Y_surv_obj <- survival::Surv(
  time = runif(n_samples, 100, 1000),
  event = sample(0:1, n_samples, replace = TRUE)
)

# Train the model
stepcox_model <- stepcox_pro(X_data, Y_surv_obj)
print(stepcox_model$finalModel)
```

`Surv` *re-export Surv from survival*

Description

re-export Surv from survival

`svm_dia` *Train a Support Vector Machine (Linear Kernel) Model for Classification*

Description

Trains a Support Vector Machine (SVM) model with a linear kernel using `caret::train` for binary classification.

Usage

```
svm_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning using `caret`'s default grid (if `TRUE`) or a fixed value (if `FALSE`).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained SVM model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
svm_model <- svm_dia(X_toy, y_toy)
print(svm_model)
```

`test_dia` *Test Data for Diagnostic Models*

Description

A test dataset for evaluating diagnostic models, with a structure identical to `train_dia`.

Usage

```
test_dia
```

Format

A data frame with rows for samples and 22 columns:

sample character. Unique identifier for each sample.

outcome integer. The binary outcome (0 or 1).

AC004637.1 numeric. Gene expression level.

AC008459.1 numeric. Gene expression level.

AC009242.1 numeric. Gene expression level.

AC016735.1 numeric. Gene expression level.

AC090125.1 numeric. Gene expression level.

AC104237.3 numeric. Gene expression level.

AC112721.2 numeric. Gene expression level.

AC246817.1 numeric. Gene expression level.

AL135841.1 numeric. Gene expression level.

AL139241.1 numeric. Gene expression level.

HYMAI numeric. Gene expression level.

KCNIP2.AS1 numeric. Gene expression level.

LINC00639 numeric. Gene expression level.
LINC00922 numeric. Gene expression level.
LINC00924 numeric. Gene expression level.
LINC00958 numeric. Gene expression level.
LINC01028 numeric. Gene expression level.
LINC01614 numeric. Gene expression level.
LINC01644 numeric. Gene expression level.
PRDM16.DT numeric. Gene expression level.

Source

Stored in `data/test_dia.rda`.

`test_pro` *Test Data for Prognostic (Survival) Models*

Description

A test dataset for evaluating prognostic models, with a structure identical to `train_pro`.

Usage

```
test_pro
```

Format

A data frame with rows for samples and 31 columns:

sample character. Unique identifier for each sample.
outcome integer. The event status (0 or 1).
time numeric. The time to event or censoring.
AC004990.1 numeric. Gene expression level.
AC055854.1 numeric. Gene expression level.
AC084212.1 numeric. Gene expression level.
AC092118.1 numeric. Gene expression level.
AC093515.1 numeric. Gene expression level.
AC104211.1 numeric. Gene expression level.
AC105046.1 numeric. Gene expression level.
AC105219.1 numeric. Gene expression level.
AC110772.2 numeric. Gene expression level.
AC133644.1 numeric. Gene expression level.
AL133467.1 numeric. Gene expression level.

AL391845.2 numeric. Gene expression level.

AL590434.1 numeric. Gene expression level.

AL603840.1 numeric. Gene expression level.

AP000851.2 numeric. Gene expression level.

AP001434.1 numeric. Gene expression level.

C9orf163 numeric. Gene expression level.

FAM153CP numeric. Gene expression level.

HOTAIR numeric. Gene expression level.

HYMAI numeric. Gene expression level.

LINC00165 numeric. Gene expression level.

LINC01028 numeric. Gene expression level.

LINC01152 numeric. Gene expression level.

LINC01497 numeric. Gene expression level.

LINC01614 numeric. Gene expression level.

LINC01929 numeric. Gene expression level.

LINC02408 numeric. Gene expression level.

SIRLNT numeric. Gene expression level.

Source

Stored in `data/test_pro.rda`.

`train_dia` *Training Data for Diagnostic Models*

Description

A training dataset for diagnostic models, containing sample IDs, binary outcomes, and gene expression features.

Usage

```
train_dia
```

Format

A data frame with rows for samples and 22 columns:

sample character. Unique identifier for each sample.

outcome integer. The binary outcome, where 1 typically represents a positive case and 0 a negative case.

AC004637.1 numeric. Gene expression level.

AC008459.1 numeric. Gene expression level.

AC009242.1 numeric. Gene expression level.

AC016735.1 numeric. Gene expression level.

AC090125.1 numeric. Gene expression level.

AC104237.3 numeric. Gene expression level.

AC112721.2 numeric. Gene expression level.

AC246817.1 numeric. Gene expression level.

AL135841.1 numeric. Gene expression level.

AL139241.1 numeric. Gene expression level.

HYMAI numeric. Gene expression level.

KCNIP2.AS1 numeric. Gene expression level.

LINC00639 numeric. Gene expression level.

LINC00922 numeric. Gene expression level.

LINC00924 numeric. Gene expression level.

LINC00958 numeric. Gene expression level.

LINC01028 numeric. Gene expression level.

LINC01614 numeric. Gene expression level.

LINC01644 numeric. Gene expression level.

PRDM16.DT numeric. Gene expression level.

Details

This dataset is used to train machine learning models for diagnosis. The column names starting with 'AC', 'AL', 'LINC', etc., are feature variables.

Source

Stored in `data/train_dia.rda`.

`train_pro` *Training Data for Prognostic (Survival) Models*

Description

A training dataset for prognostic models, containing sample IDs, survival outcomes (time and event status), and gene expression features.

Usage

```
train_pro
```

Format

A data frame with rows for samples and 31 columns:

sample character. Unique identifier for each sample.
outcome integer. The event status, where 1 indicates an event occurred and 0 indicates censoring.
time numeric. The time to event or censoring.
AC004990.1 numeric. Gene expression level.
AC055854.1 numeric. Gene expression level.
AC084212.1 numeric. Gene expression level.
AC092118.1 numeric. Gene expression level.
AC093515.1 numeric. Gene expression level.
AC104211.1 numeric. Gene expression level.
AC105046.1 numeric. Gene expression level.
AC105219.1 numeric. Gene expression level.
AC110772.2 numeric. Gene expression level.
AC133644.1 numeric. Gene expression level.
AL133467.1 numeric. Gene expression level.
AL391845.2 numeric. Gene expression level.
AL590434.1 numeric. Gene expression level.
AL603840.1 numeric. Gene expression level.
AP000851.2 numeric. Gene expression level.
AP001434.1 numeric. Gene expression level.
C9orf163 numeric. Gene expression level.
FAM153CP numeric. Gene expression level.
HOTAIR numeric. Gene expression level.
HYMAI numeric. Gene expression level.
LINC00165 numeric. Gene expression level.
LINC01028 numeric. Gene expression level.
LINC01152 numeric. Gene expression level.
LINC01497 numeric. Gene expression level.
LINC01614 numeric. Gene expression level.
LINC01929 numeric. Gene expression level.
LINC02408 numeric. Gene expression level.
SIRLNT numeric. Gene expression level.

Details

This dataset is used to train machine learning models for prognosis. The features are typically gene expression values.

Source

Stored in `data/train_pro.rda`.

voting_dia *Train a Voting Ensemble Diagnostic Model*

Description

Implements a Voting ensemble, combining predictions from multiple base models through soft or hard voting.

Usage

```
voting_dia(
  results_all_models,
  data,
  type = c("soft", "hard"),
  weight_metric = "AUROC",
  top = 5,
  seed = 789,
  threshold_choices = "f1",
  positive_label_value = 1,
  negative_label_value = 0,
  new_positive_label = "Positive",
  new_negative_label = "Negative"
)
```

Arguments

`results_all_models`
: A list of results from `models_dia()`, containing trained base model objects and their evaluation metrics.

`data`
: A data frame where the first column is the sample ID, the second is the outcome label, and subsequent columns are features. Used for evaluation.

`type`
: A character string, "soft" for weighted average of probabilities or "hard" for majority class voting.

`weight_metric`
: A character string, the metric to use for weighting base models in soft voting (e.g., "AUROC", "F1"). Ignored for hard voting.

`top`
: An integer, the number of top-performing base models (ranked by `weight_metric`) to include in the ensemble.

`seed`
: An integer, for reproducibility.

`threshold_choices`
: A character string (e.g., "f1", "youden", "default") or a numeric value (0-1) for determining the evaluation threshold for the ensemble.

`positive_label_value`
: A numeric or character value in the raw data representing the positive class.

`negative_label_value`
: A numeric or character value in the raw data representing the negative class.

new_positive_label
: A character string, the desired factor level name for the positive class (e.g., "Positive").

new_negative_label
: A character string, the desired factor level name for the negative class (e.g., "Negative").

Value

A list containing the model_object, sample_score, and evaluation_metrics.

See Also

models_dia, evaluate_model_dia

Examples

```
# 1. Initialize the modeling system
initialize_modeling_system_dia()

# 2. Create a toy dataset for demonstration
set.seed(42)
data_toy <- data.frame(
  ID = paste0("Sample", 1:60),
  Status = sample(c(0, 1), 60, replace = TRUE),
  Feat1 = rnorm(60),
  Feat2 = runif(60)
)

# 3. Generate mock base model results (as if from models_dia)
base_model_results <- models_dia(
  data = data_toy,
  model = c("rf", "lasso"),
  seed = 123
)

# 4. Run the soft voting ensemble
soft_voting_results <- voting_dia(
  results_all_models = base_model_results,
  data = data_toy,
  type = "soft",
  weight_metric = "AUROC",
  top = 2,
  threshold_choices = "f1"
)
print_model_summary_dia("Soft Voting", soft_voting_results)
```

xb_dia *Train an XGBoost Tree Model for Classification*

Description

Trains an Extreme Gradient Boosting (XGBoost) model using `caret::train` for binary classification.

Usage

```
xb_dia(X, y, tune = FALSE, cv_folds = 5)
```

Arguments

`X`	A data frame of features.
`y`	A factor vector of class labels.
`tune`	Logical, whether to perform hyperparameter tuning using `caret`'s default grid (if `TRUE`) or use fixed values (if `FALSE`).
`cv_folds`	An integer, the number of cross-validation folds for `caret`.

Value

A `caret::train` object representing the trained XGBoost model.

Examples

```
set.seed(42)
n_obs <- 50
X_toy <- data.frame(
  FeatureA = rnorm(n_obs),
  FeatureB = runif(n_obs, 0, 100)
)
y_toy <- factor(sample(c("Control", "Case"), n_obs, replace = TRUE),
                levels = c("Control", "Case"))

# Train the model
xb_model <- xb_dia(X_toy, y_toy)
print(xb_model)
```

Index