Package: evoFE (via r-universe)

June 10, 2026

Type Package

Title Evolutionary Feature Engineering

Version 0.2.0

Description Automates feature engineering using evolutionary algorithms inspired by genetic programming. Starting from raw input features, the package evolves candidate transformation recipes through selection, crossover, and mutation, evaluating fitness via cross-validation or train/validation splits with gradient-boosted tree models ('LightGBM' or 'XGBoost'). Built-in transformers include arithmetic, logarithmic, and power operations, interaction terms, target encoding, quantile and log-based binning, principal component analysis, truncated singular value decomposition, Uniform Manifold Approximation and Projection (UMAP) dimensionality reduction, and minimum spanning tree (MST) graph-based clustering. The evolutionary search yields an optimised feature recipe that can be applied to new data for prediction. Methods are described in McInnes et al. (2018) <doi:10.21105/joss.00861>, Ke et al. (2017) <https://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-framework>, Chen and Guestrin (2016) <doi:10.1145/2939672.2939785>, Gagolewski (2021) <doi:10.1016/j.softx.2021.100722>, Gagolewski (2026) <doi:10.32614/CRAN.package.lumbermark>, and Gagolewski (2026) <doi:10.32614/CRAN.package.deadwood>.

License MIT + file LICENSE

Encoding UTF-8

Imports data.table, lightgbm, xgboost, stats, digest, uwot, quitefastmst, genieclust, ParamHelpers, lhs, mlrMBO, smoof

Suggests RhpcBLASctl, testthat, knitr, rmarkdown, lumbermark, deadwood, mlr, DiceKriging, randomForest, emoa

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

RoxygenNote 7.3.2

Config/pak/sysreqs libgdal-dev gdal-bin libgeos-dev libglu1-mesa-dev libgmp3-dev make libgsl0-dev jags libicu-dev libxml2-dev libmpfr-dev libopenmpi-dev libproj-dev

Repository https://tanopereira.r-universe.dev

Date/Publication 2026-06-09 17:59:39 UTC

RemoteUrl https://github.com/tanopereira/evofe

RemoteRef HEAD

RemoteSha 907e5038a8f76764779ca85413c4c8a580d09401

Contents

apply_gene *Apply a single gene to a dataset*

Description

Apply a single gene to a dataset

Usage

```
apply_gene(
  gene,
  train_data,
  val_data = NULL,
  target_col = NULL,
  state_cache = NULL,
  data_hash = NULL
)
```

Arguments

gene	A gene list representing a feature transformation.
train_data	A data.frame or data.table representing the training data.
val_data	Optional validation data.frame or data.table.
target_col	Name of the target column.
state_cache	Optional environment to cache full-dataset fitted states of stateful transformers.
data_hash	Optional pre-computed xxhash64 digest of the target column, to avoid redundant hashing when applying multiple genes.

Value

A list with three elements: `train` (the modified training `data.table` with the new gene column appended), `val` (the modified validation `data.table` or `NULL`), and `gene` (the gene list, with its `state` element populated if the transformer is stateful).

apply_individual *Apply an entire individual's recipe to data*

Description

Apply an entire individual's recipe to data

Usage

```
apply_individual(
  ind,
  train_data,
  val_data = NULL,
  target_col = NULL,
  state_cache = NULL,
  allow_prune = TRUE
)
```

Arguments

ind	An evo_individual object.
train_data	A data.frame or data.table representing the training data.
val_data	Optional validation data.frame or data.table.
target_col	Name of the target column.
state_cache	Optional environment to cache full-dataset fitted states of stateful transformers.
allow_prune	Logical. If TRUE, genes that fail application are skipped instead of failing the entire individual.

Value

A list with three elements: `train` (the transformed training `data.table` with all gene columns applied), `val` (the transformed validation `data.table` or `NULL`), and `ind` (the updated `evo_individual` whose genes now carry fitted states).

---

compute_ts_refinement *Temperature Scaled Refinement Metric*

Description

Computes the Temperature Scaled Refinement (TS-Refinement) metric for binary or multiclass classification. The metric finds the temperature T that minimizes the Laplace-smoothed log-loss of the temperature-scaled prediction margins (logits).

Usage

```
compute_ts_refinement(
  y_true,
  y_pred,
  task = "classification",
  num_class = NULL,
  alpha = 1,
  is_logits = FALSE
)
```

Arguments

y_true	Numeric vector of true labels (0/1 for classification, or 0 to C-1 for multiclass classification).
y_pred	Numeric vector or matrix of predicted probabilities (or logits, if is_logits = TRUE).
task	Character. Either "classification" (binary) or "multiclass".
num_class	Integer. Number of classes (required for multiclass).
alpha	Numeric. Laplace smoothing parameter (default is 1).
is_logits	Logical. If TRUE, the input predictions y_pred are treated directly as prediction margins (logits). If FALSE, they are treated as probabilities and converted to logits.

Value

Numeric. The minimized smoothed log-loss.

---

create_gene *Create a single gene*

Description

Create a single gene

Usage

```
create_gene(transformer_name, input_cols)
```

Arguments

transformer_name	Name of the transformer
input_cols	Vector of input column names

Value

A gene list with elements transformer_name, input_cols, params (transformer-specific parameters), state (NULL until fitted), and output_col (auto-generated column name).

create_individual *Create an individual*

Description

Create an individual

Usage

```
create_individual(
  genes = list(),
  numeric_cols = character(0),
  categorical_cols = character(0)
)
```

Arguments

genes	List of genes
numeric_cols	Vector of numeric column names
categorical_cols	Vector of categorical column names

Value

An evo_individual S3 object: a list with elements genes (topologically sorted), numeric_cols, categorical_cols, and fitness (initialised to NA_real_).

create_transformer *Create a transformer definition*

Description

Create a transformer definition

Usage

```
create_transformer(
  name,
  type,
  input_type = "numeric",
  output_type = "numeric",
  fit_func = NULL,
  apply_func,
  name_generator,
  allow_replace = FALSE
)
```

Arguments

name	Transformer name
type	Type: "unary", "binary", "supervised_unary"
input_type	Type of input: "numeric" or "categorical"
output_type	Type of output: "numeric" or "categorical"
fit_func	function(data, input_cols, target_col = NULL) returning state
apply_func	function(data, input_cols, state = NULL) returning new column vector
name_generator	function(input_cols) returning output column name
allow_replace	Logical. Whether column sampling allows replacement.

Value

An evo_transformer S3 object: a list with elements name, type, input_type, output_type, fit_func, apply_func, name_generator, and allow_replace.

Examples

```
# Define a transformer that adds a constant value of 10 to a variable
add_ten_trans <- create_transformer(
  name = "add_ten",
  type = "unary",
  input_type = "numeric",
  apply_func = function(data, gene, state = NULL) {
    data[[gene$input_cols[1]]] + 10
  },
  name_generator = function(gene) paste0("add10_", gene$input_cols[1])
)
print(add_ten_trans)
```

---

crossover *Crossover two individuals*

Description

Crossover two individuals

Usage

```
crossover(ind1, ind2, verbose = FALSE)
```

Arguments

ind1	Parent 1
ind2	Parent 2
verbose	Logical. Whether to print crossover details.

Value

An `evo_individual` child created by randomly sampling genes from both parents with duplicate gene outputs removed.

`evaluate_fitness` *Evaluate the fitness of an individual*

Description

Evaluate the fitness of an individual

Usage

```
evaluate_fitness(
  ind,
  data,
  target_col,
  task = "classification",
  cv_folds = 3,
  evaluation_strategy = "cv",
  split_ids = NULL,
  shared_splits = NULL,
  evaluator = "lightgbm",
  fold_ids = NULL,
  shared_folds = NULL,
  shared_full = NULL,
  state_cache = NULL,
  threads = 2,
  metric = "default",
  verbose = FALSE,
  allow_prune = TRUE,
  ...
)
```

Arguments

`ind`	An evo_individual object.
`data`	A data.frame or data.table containing the dataset.
`target_col`	Name of the target column.
`task`	"classification" or "regression".
`cv_folds`	Number of cross-validation folds.
`evaluation_strategy`	Character string, either "cv" (cross-validation) or "split" (train/validation split).
`split_ids`	Optional vector of pre-defined split assignments (e.g. "train", "val", "holdout").
`shared_splits`	Optional list of shared data.table splits for in-place caching.

evaluator	The ML model to use ("lightgbm", "xgboost", "catboost", or a custom registered evaluator name).
fold_ids	Optional vector of pre-defined fold assignments.
shared_folds	Optional list of shared data.table CV folds for in-place caching.
shared_full	Optional data.table of the full dataset for in-place caching.
state_cache	Optional environment to cache full-dataset fitted states of stateful transformers.
threads	Number of threads to use for parallel execution (default 2)
metric	The metric to optimize ("default", "auc", "f1", "mae", or a custom function).
verbose	Logical indicating if progress should be printed.
allow_prune	Logical. If TRUE, genes that fail application are skipped instead of failing the entire individual.
...	Additional arguments passed to the underlying evaluator training functions.

Value

The input evo_individual with its fitness field set to the computed score (higher is better), importances set to a named numeric vector of feature importances, holdout_fitness set to NULL, and genes updated with fitted transformer states.

---

evo_evaluators *Global environment for registered model evaluators*

Description

Global environment for registered model evaluators

Usage

```
evo_evaluators
```

---

evo_transformers *Built-in feature transformers*

Description

An environment containing default transformer definitions available for feature engineering.

Usage

```
evo_transformers
```

Value

A named list of evo_transformer objects, each defining a feature transformation (e.g. log, pca, target_encode).

`evolve_features` *Run evolutionary feature engineering*

Description

Run evolutionary feature engineering

Usage

```
evolve_features(
  data,
  target_col,
  task = "classification",
  generations = 10,
  pop_size = 10,
  cv_folds = 3,
  evaluation_strategy = "cv",
  split_ratio = c(0.6, 0.2, 0.2),
  split_ids = NULL,
  early_stopping_rounds = 3,
  evaluator = "lightgbm",
  dynamic_population = TRUE,
  dynamic_population_growth_rate = 1.5,
  dynamic_population_decay_rate = 0.7,
  crossover_type = "both",
  threads = 2,
  max_clustering_size = 5000,
  verbose = TRUE,
  metric = "default",
  model_all_final_genes = FALSE,
  model_all_historical_genes = FALSE,
  ...
)
```

Arguments

`data`	A data.frame or data.table
`target_col`	Name of the target column
`task`	"classification" or "regression"
`generations`	Number of generations (max iterations)
`pop_size`	Population size
`cv_folds`	Number of cross-validation folds
`evaluation_strategy`	"cv" or "split". Strategy to evaluate candidate recipes.

split_ratio A numeric vector of length 2 or 3 defining train/validation/holdout proportions (e.g. c(0.6, 0.2, 0.2)).

split_ids An optional character vector of split assignments (e.g. "train", "val", "holdout").

early_stopping_rounds
Stop if fitness doesn't improve for this many generations

evaluator The ML model to use ("lightgbm", "xgboost", "catboost", or a custom registered evaluator name).

dynamic_population
Logical. If TRUE, population expands dynamically during stagnation.

dynamic_population_growth_rate
Growth rate multiplier for population expansion during stagnation (default 1.5).

dynamic_population_decay_rate
Decay rate multiplier for population contraction back to baseline (default 0.7).

crossover_type Crossover type: "both" (default, 50% random / 50% union), "random", or "union"

threads Number of threads to use for parallel execution (default 2)

max_clustering_size
Maximum unique training rows to cluster (default 5000, 0/NULL for unlimited)

verbose Logical. If TRUE, prints progress.

metric The metric to optimize ("default", "auc", "f1", "mae", or a custom function).

model_all_final_genes
Logical. If TRUE, the final model is trained using the union of all unique genes evolved in the final population, rather than only the best individual's genes.

model_all_historical_genes
Logical. If TRUE, the final model is trained using the union of all unique genes evolved across all generations, rather than only the best individual's genes.

... Additional arguments passed to the underlying evaluator training functions.

Value

An evo_recipe S3 object: a list with elements best_individual (the top-scoring evo_individual), history (list of all evaluated individuals across generations), task, best_model (the trained model object), evaluator, and classes (class levels for multiclass tasks, otherwise NULL).

Examples

```
# Quick classification example using mtcars
data(mtcars)
df <- mtcars
df$am <- as.integer(df$am)

set.seed(42)
recipe <- evolve_features(
  data = df,
  target_col = "am",
  task = "classification",
  evaluator = "xgboost",
```

```
  generations = 2,
  pop_size = 2,
  cv_folds = 2,
  verbose = FALSE
)
print(recipe)
```

`gene_to_formula` *Convert a gene to a formula string*

Description

Convert a gene to a formula string

Usage

```
gene_to_formula(gene)
```

Arguments

`gene`	A gene list

Value

A character string representing the gene as a human-readable formula, e.g. `"log(col1)"` or `"pca2(col1, col2)"`.

`gene_to_state_formula` *Convert a gene to a formula string for state caching (ignoring component index)*

Description

Convert a gene to a formula string for state caching (ignoring component index)

Usage

```
gene_to_state_formula(gene)
```

Arguments

`gene`	A gene list

Value

A character string representing the gene formula suitable for state caching. For multi-component transformers (PCA, SVD, UMAP) the component index is omitted so that all components share one cache key.

individual_to_recipe_string *Convert an individual to a recipe string of formulas*

Description

Convert an individual to a recipe string of formulas

Usage

```
individual_to_recipe_string(ind)
```

Arguments

ind	An evo_individual

Value

A character string listing all gene formulas in bracket notation, e.g. `"[log(x), sqrt(y)]"`, or `"[Original features only]"` when the individual has no genes.

initialize_population *Initialize a population*

Description

Initialize a population

Usage

```
initialize_population(
  pop_size,
  numeric_cols,
  categorical_cols,
  initial_genes = 2,
  task = "classification",
  importances = NULL
)
```

Arguments

pop_size	Population size.
numeric_cols	Vector of numeric column names.
categorical_cols	Vector of categorical column names.
initial_genes	Number of initial genes per individual.
task	Task type ("classification", "regression", or "multiclass").
importances	Optional numeric vector of feature importances.

Value

A list of evo_individual objects of length pop_size. The first individual is a baseline with no genes; the remaining individuals each carry initial_genes randomly generated genes.

---

make_tunable *Create a Tunable Evaluator from a Registered Base Model*

Description

Wraps an existing registered model evaluator in a Bayesian Optimization tuning loop using the **mlrMBO** framework. It automatically generates a parameter space, constructs a cross-validation or split-validation objective function, searches for the optimal hyperparameters, and registers the tuned evaluator.

Usage

```
make_tunable(
  base_model_name,
  param_ranges,
  tuner_name = paste0(base_model_name, "_mbo")
)
```

Arguments

base_model_name	Character. Name of the registered base evaluator (e.g., "xgboost", "lightgbm").
param_ranges	List. A nested list defining the parameter names, types, and bounds/values. Each parameter definition must be a list containing: type Character: "numeric", "integer", or "discrete". lower Numeric/Integer: Lower bound of the search space (required for "numeric" and "integer"). upper Numeric/Integer: Upper bound of the search space (required for "numeric" and "integer"). values Vector: Set of valid values (required for "discrete").
tuner_name	Character. The name under which to register the tuned evaluator. Defaults to paste0(base_model_name, "_mbo").

Details

The tuning loop uses a Latin Hypercube Design (LHS) for the initial parameters layout. It attempts to use a Kriging (Gaussian Process) surrogate model by default and automatically falls back to a Random Forest surrogate model if numerical singularities are encountered (which is common on small datasets).

Value

Invisibly returns `NULL`. Registers the tuned evaluator in the global `evo_evaluators` environment.

Examples

```
## Not run:
# 1. Register a simple mock evaluator
register_evaluator(
  "mock_base",
 train_func = function(x_train, y_train, x_val = NULL, y_val = NULL, task = "regression", ...) {
    args <- list(...)
    val_score <- 100 - abs(args$param_a - 4.5)
    list(
      model = list(args = args, val_score = val_score),
      predictions = if (!is.null(x_val)) rep(val_score, nrow(x_val)) else NULL
    )
  },
  predict_func = function(model, x_new, task, ...) {
    rep(model$val_score, nrow(x_new))
  }
)

# 2. Make it tunable
param_ranges <- list(
  param_a = list(type = "numeric", lower = 1.0, upper = 8.0)
)
make_tunable("mock_base", param_ranges, tuner_name = "mock_tuned")

# 3. Train the tuned model on mock data
x_train <- matrix(rnorm(20), ncol = 2)
colnames(x_train) <- c("x1", "x2")
y_train <- rnorm(10)
x_val <- matrix(rnorm(10), ncol = 2)
y_val <- rnorm(5)

fit <- train_model(
  x_train, y_train, x_val = x_val, y_val = y_val, task = "regression",
  evaluator = "mock_tuned", mbo_iters = 3, mbo_init_design = 5, mbo_folds = 2
)
print(fit$best_params)

## End(Not run)
```

`mutate` *Mutate an individual*

Description

Mutate an individual

Usage

```
mutate(
  ind,
  verbose = FALSE,
  force_add = FALSE,
  importances = numeric(0),
  temperature = 1,
  task = "classification",
  tested_gene_outputs = NULL
)
```

Arguments

`ind`	An evo_individual.
`verbose`	Logical. Whether to print mutation details.
`force_add`	Logical. If TRUE, forces adding a new gene.
`importances`	A numeric vector of feature importances.
`temperature`	A numeric temperature value controlling selection weights.
`task`	The task type ("classification", "regression", or "multiclass")
`tested_gene_outputs`	Character vector of gene output names that have been evaluated in a previous generation and are safe for chaining. When NULL (default), all existing gene outputs are available. Pass character(0) to block all chaining (e.g. during initialization).

Value

An `evo_individual` with the mutation applied (gene added, removed, or modified) and `fitness` reset to `NA_real_`.

`plot.evo_recipe` *Plot an evo_recipe object*

Description

Plots either the fitness trajectory over generations or the feature importances of the best individual.

Usage

```
## S3 method for class 'evo_recipe'
plot(x, type = "fitness", ...)
```

Arguments

`x`	An `evo_recipe` object.
`type`	Character string, either `"fitness"` (default) to plot the fitness trajectory, or `"importance"` to plot a bar chart of the top feature importances of the winning model.
`...`	Additional arguments passed to `plot` or `barplot`.

Examples

```
data(mtcars)
df <- mtcars
df$am <- as.integer(df$am)

recipe <- evolve_features(
  data = df,
  target_col = "am",
  task = "classification",
  evaluator = "xgboost",
  generations = 2,
  pop_size = 2,
  cv_folds = 2,
  seed = 42,
  verbose = FALSE
)

# Plot the fitness curve
plot(recipe, type = "fitness")

# Plot feature importances
plot(recipe, type = "importance")
```

predict.evo_recipe *Apply feature engineering recipe to new data*

Description

Apply feature engineering recipe to new data

Usage

```
## S3 method for class 'evo_recipe'
predict(object, newdata, ...)
```

Arguments

object	An evo_recipe object
newdata	A data.frame or data.table
...	Additional arguments

Value

A data.table containing the engineered feature columns (original plus all gene-derived columns) for newdata, ready for downstream modelling.

Examples

```
data(mtcars)
df <- mtcars
df$am <- as.integer(df$am)

recipe <- evolve_features(
  data = df,
  target_col = "am",
  task = "classification",
  evaluator = "xgboost",
  generations = 2,
  pop_size = 2,
  cv_folds = 2,
  seed = 42,
  verbose = FALSE
)

# Extract engineered features
engineered_features <- predict(recipe, df[1:5, ])
print(engineered_features)
```

predict_model *Predict target values using the fully evolved model*

Description

Predict target values using the fully evolved model

Usage

```
predict_model(object, newdata, ...)
```

Arguments

object	An evo_recipe object containing the trained model and best individual
newdata	A data.frame or data.table to make predictions on
...	Additional arguments (currently unused)

Value

For binary classification and regression tasks a numeric vector of predictions. For multiclass tasks a numeric matrix with one column per class (columns named after class levels).

Examples

```
data(mtcars)
df <- mtcars
df$am <- as.integer(df$am)

recipe <- evolve_features(
  data = df,
  target_col = "am",
  task = "classification",
  evaluator = "xgboost",
  generations = 2,
  pop_size = 2,
  cv_folds = 2,
  seed = 42,
  verbose = FALSE
)

# Get model predictions
predictions <- predict_model(recipe, df[1:5, ])
print(predictions)
```

print.evo_recipe *Print an evo_recipe object*

Description

Prints a human-readable summary of the evolutionary feature engineering recipe.

Usage

```
## S3 method for class 'evo_recipe'
print(x, ...)
```

Arguments

x	An evo_recipe object.
...	Additional arguments (currently unused).

print.summary_evo_recipe *Print summary of an evo_recipe object*

Description

Prints the summary details of the evolutionary feature engineering recipe.

Usage

```
## S3 method for class 'summary_evo_recipe'
print(x, ...)
```

Arguments

x	A summary_evo_recipe object.
...	Additional arguments (currently unused).

`register_evaluator` *Register a model evaluator*

Description

Register a model evaluator

Usage

```
register_evaluator(name, train_func, predict_func, base_evaluator = NULL)
```

Arguments

`name`	Name of the evaluator.
`train_func`	Function to train the model. Must accept `x_train`, `y_train`, `x_val`, `task`, `threads`, `num_class`, and any additional parameters, and return a list with `model`, `predictions`, and `importances`.
`predict_func`	Function to make predictions. Must accept `model`, `x_new`, `task`, and any additional parameters, and return a vector or matrix of predictions.
`base_evaluator`	Optional character name of the base registered model.

Examples

```
# Register a simple mock evaluator
register_evaluator(
  "mock_eval",
  train_func = function(x_train, y_train, x_val = NULL, task = "regression", ...) {
    list(
      model = list(weights = colMeans(x_train)),
      predictions = if (!is.null(x_val)) rowMeans(x_val) else NULL,
      importances = stats::setNames(rep(1, ncol(x_train)), colnames(x_train))
    )
  },
  predict_func = function(model, x_new, task, ...) {
    rowMeans(x_new)
  }
)

# Verify it is registered
exists("mock_eval", envir = evo_evaluators)
```

register_transformer *Register a custom feature transformer*

Description

Adds a user-defined feature transformer to the available pool for feature evolution.

Usage

```
register_transformer(name, transformer)
```

Arguments

name	Unique character string naming the transformer.
transformer	An object of class evo_transformer created via create_transformer.

Examples

```
# Create a custom transformer
add_ten_trans <- create_transformer(
  name = "add_ten",
  type = "unary",
  input_type = "numeric",
  apply_func = function(data, gene, state = NULL) {
    data[[gene$input_cols[1]]] + 10
  },
  name_generator = function(gene) paste0("add10_", gene$input_cols[1])
)

# Register it
register_transformer("add_ten", add_ten_trans)

# Verify it is registered
exists("add_ten", envir = evo_transformers)
```

summary.evo_recipe *Summary of an evo_recipe object*

Description

Computes and formats a detailed summary of the evolutionary feature engineering recipe.

Usage

```
## S3 method for class 'evo_recipe'
summary(object, ...)
```

Arguments

object	An evo_recipe object.
...	Additional arguments (currently unused).

Examples

```
data(mtcars)
df <- mtcars
df$am <- as.integer(df$am)

recipe <- evolve_features(
  data = df,
  target_col = "am",
  task = "classification",
  evaluator = "xgboost",
  generations = 2,
  pop_size = 2,
  cv_folds = 2,
  seed = 42,
  verbose = FALSE
)

# Print the recipe overview
print(recipe)

# Inspect a detailed summary
recipe_summary <- summary(recipe)
print(recipe_summary)
```

---

union_crossover *Union Crossover of two individuals*

Description

Union Crossover of two individuals

Usage

```
union_crossover(ind1, ind2, verbose = FALSE)
```

Arguments

ind1	Parent 1
ind2	Parent 2
verbose	Logical. Whether to print crossover details.

Value

An `evo_individual` child created by taking the union of all genes from both parents with duplicate gene outputs removed.

Index