Package: covalchemy (via r-universe)

November 25, 2024

Title Constructing Joint Distributions with Control Over Statistical Properties

Version 1.0.0

Description Synthesizing joint distributions from marginal densities, focusing on controlling key statistical properties such as correlation for continuous data, mutual information for categorical data, and inducing Simpson's Paradox. Generate datasets with specified correlation structures for continuous variables, adjust mutual information between categorical variables, and manipulate subgroup correlations to intentionally create Simpson's Paradox. Joe (1997) <doi:10.1201/b13150> Sklar (1959) <https://en.wikipedia.org/wiki/Sklar%27s_theorem>.

Encoding UTF-8

Imports dplyr, ggplot2, mvtnorm, interp, clue, ggExtra, gridExtra, DescTools, MASS

Depends R (>= 3.5.0)

License GPL-3

BugReports https://github.com/namanlab/covalchemy/issues

URL https://github.com/namanlab/covalchemy

RoxygenNote 7.3.2

Config/testthat/edition 3

NeedsCompilation no

Author Naman Agrawal [aut, cre]

Maintainer Naman Agrawal <naman.agr03@gmail.com>

Repository CRAN

Date/Publication 2024-11-21 17:10:11 UTC

Config/pak/sysreqs make libssl-dev libx11-dev zlib1g-dev

Contents

augment_matrix_random_block . . . 2
calculate_tv_distance_empirical . . . 3
entropy_pair . . . 4
gaussian_copula_two_vars . . . 5
genCDFInv_akima . . . 6
genCDFInv_linear . . . 7
genCDFInv_poly . . . 8
genCDFInv_quantile . . . 9
generate_gaussian_copula_samples . . . 10
generate_t_copula_samples . . . 11
gen_number_1 . . . 12
gen_number_max . . . 13
gen_number_min . . . 14
get_mutual_information . . . 15
get_optimal_grid . . . 16
get_simpsons_paradox_c . . . 17
get_simpsons_paradox_d . . . 18
get_target_corr . . . 20
get_target_entropy . . . 21
log_odds_dc . . . 22
objective_function_SL . . . 23
plot_log_odds . . . 25
simulated_annealing_MI . . . 26
simulated_annealing_SL . . . 27
sinkhorn_algorithm . . . 28
softmax . . . 29
t_copula_two_vars . . . 30

Index **31**

`augment_matrix_random_block` *Augment Matrix with Random 2x2 Block Adjustment*

Description

This function selects a random 2x2 block of values in the input matrix `table` and modifies them based on the specified delta. It checks certain conditions before applying the modifications to the selected block. The process repeats until a valid block is found or a maximum of 100 iterations is reached.

Usage

```
augment_matrix_random_block(table, delta)
```

Arguments

table	A matrix (numeric) to which the random block adjustment will be applied.
delta	A numeric value that determines the magnitude of the adjustment. If positive, values are subtracted from the block; if negative, values are added.

Value

A matrix (numeric) with the adjusted 2x2 block.

Examples

```
table <- matrix(1:9, 3, 3)
augment_matrix_random_block(table, 1)
```

---

calculate_tv_distance_empirical

Calculate Total Variation (TV) Distance Empirically

Description

This function calculates the Total Variation (TV) distance between the empirical cumulative distribution functions (ECDFs) of two datasets: original data and generated data. The TV distance is defined as half the sum of the absolute differences between the two CDFs at each point in the domain.

Usage

```
calculate_tv_distance_empirical(original_data, generated_data)
```

Arguments

original_data	A numeric vector of the original data.
generated_data	A numeric vector of the generated data.

Value

A numeric value representing the Total Variation distance between the empirical CDFs of the original and generated data.

Examples

```
# Test Case 1: Data from similar distributions
original_data <- rnorm(1000, mean = 0, sd = 1)  # Normal distribution (mean = 0, sd = 1)
generated_data <- rnorm(1000, mean = 0, sd = 1)  # Similar normal distribution
tv_distance <- calculate_tv_distance_empirical(original_data, generated_data)
print(tv_distance)  # Expected to be close to 0, as both datasets are similar

# Test Case 2: Data from different distributions
original_data <- rnorm(1000, mean = 0, sd = 1)  # Normal distribution (mean = 0, sd = 1)
generated_data <- rnorm(1000, mean = 5, sd = 2)  # Different normal distribution
tv_distance <- calculate_tv_distance_empirical(original_data, generated_data)
print(tv_distance)  # Expected to be larger, as the datasets are quite different
```

---

entropy_pair *Calculate Entropy of a Pair*

Description

This function calculates the entropy of a pair of variables (or a pairwise contingency table) based on the probability distribution of their joint occurrences.

Usage

```
entropy_pair(table)
```

Arguments

table	A numeric vector or matrix. A contingency table or frequency table of a pair of variables.

Details

The entropy is calculated using the formula: [H(X, Y) = - sum p(x, y) * log_2(p(x, y))] where \(p(x, y) \) is the probability of observing the pair \((x, y) \) from the table.

Value

A numeric value representing the entropy of the pair.

Examples

```
# Example usage with a simple contingency table:
pair_table <- table(c(1, 2, 2, 3), c(1, 1, 2, 2))
entropy_pair(pair_table)
```

`gaussian_copula_two_vars` *Generate Gaussian Copula Samples for Two Variables*

Description

This function generates samples from a Gaussian copula with two variables, given the specified correlation coefficient between the variables.

Usage

```
gaussian_copula_two_vars(n, p)
```

Arguments

`n`	Integer. The number of samples to generate.
`p`	Numeric. The correlation coefficient (ρ) between the two variables. Must be in the range `[-1, 1]`.

Details

The function internally constructs a correlation matrix for two variables:

$$\rho_matrix = \begin{bmatrix} 1 & p \\ p & 1 \end{bmatrix}$$

It then calls `generate_gaussian_copula_samples` to generate samples.

Value

A matrix of size `n x 2`, where each row represents a sample, and each column corresponds to one of the two variables. The values are uniformly distributed in `[0, 1]`.

See Also

`generate_gaussian_copula_samples`

Examples

```
# Example usage:
samples <- gaussian_copula_two_vars(n = 1000, p = 0.7)
head(samples)
```

genCDFInv_akima *Generate an Inverse CDF Function Using Akima Spline Interpolation*

Description

This function creates an inverse cumulative distribution function (CDF) for a given dataset using Akima spline interpolation. The resulting function maps probabilities (in the range `[0, 1]`) to values in the dataset.

Usage

```
genCDFInv_akima(X)
```

Arguments

`X`	A numeric vector. The dataset for which the inverse CDF is to be created.

Details

The function works as follows:

1. Computes the empirical CDF (ECDF) of the dataset.
2. Extracts the sorted ECDF values for the dataset.
3. Sorts the original data values.
4. Uses the `aspline` function to create a spline interpolation mapping probabilities to dataset values.

The resulting function leverages Akima splines, which are smooth and flexible for interpolating data.

Value

A function that takes a single argument, `p`, a numeric vector of probabilities in `[0, 1]`, and returns the corresponding values interpolated from the dataset using Akima splines.

See Also

`ecdf`, `aspline`

Examples

```
# Example usage:
library(interp)
data <- c(1, 2, 3, 4, 5)
inv_cdf <- genCDFInv_akima(data)
inv_cdf(c(0.1, 0.5, 0.9))  # Compute interpolated values for given probabilities
```

genCDFInv_linear *Generate an Inverse CDF Function Using Linear Interpolation*

Description

This function creates an inverse cumulative distribution function (CDF) for a given dataset using linear interpolation. The resulting function maps probabilities (in the range `[0, 1]`) to values in the dataset.

Usage

```
genCDFInv_linear(X)
```

Arguments

`X`	A numeric vector. The dataset for which the inverse CDF is to be created.

Details

The function works as follows:

1. Computes the empirical CDF (ECDF) of the dataset.
2. Extracts the sorted ECDF values for the dataset.
3. Sorts the original data values.
4. Uses `approxfun` to create a linear interpolation function mapping probabilities to dataset values.

The resulting function can handle probabilities outside `[0, 1]` using the `rule = 2` parameter in `approxfun`, which extrapolates based on the nearest data points.

Value

A function that takes a single argument, `p`, a numeric vector of probabilities in `[0, 1]`, and returns the corresponding values interpolated from the dataset.

See Also

`ecdf`, `approxfun`

Examples

```
# Example usage:
data <- c(1, 2, 3, 4, 5)
inv_cdf <- genCDFInv_linear(data)
inv_cdf(c(0.1, 0.5, 0.9))  # Compute the interpolated values for given probabilities
```

genCDFInv_poly *Generate an Inverse CDF Function Using Polynomial Regression*

Description

This function creates an inverse cumulative distribution function (CDF) for a given dataset using polynomial regression. The resulting function maps probabilities (in the range `[0, 1]`) to values in the dataset.

Usage

```
genCDFInv_poly(data, degree)
```

Arguments

`data`	A numeric vector. The dataset for which the inverse CDF is to be created.
`degree`	An integer. The degree of the polynomial to fit in the regression.

Details

The function works as follows:

1. Sorts the dataset and computes the empirical CDF (ECDF) of the data.
2. Fits a polynomial regression to model the relationship between ECDF values and the sorted dataset.
3. Uses the fitted polynomial model to predict the inverse CDF for given probabilities.

The degree of the polynomial can be specified to control the flexibility of the regression model.

Value

A function that takes a single argument, `y`, a numeric vector of probabilities in `[0, 1]`, and returns the corresponding values predicted by the polynomial regression.

See Also

`lm`, `poly`, `ecdf`

Examples

```
# Example usage:
data <- c(1, 2, 3, 4, 5)
inv_cdf <- genCDFInv_poly(data, degree = 2)
inv_cdf(c(0.1, 0.5, 0.9))  # Compute predicted values for given probabilities
```

genCDFInv_quantile *Generate an Inverse CDF Function Using Quantiles*

Description

This function creates an inverse cumulative distribution function (CDF) for a given dataset using quantiles. The resulting function maps probabilities (in the range `[0, 1]`) to quantiles of the data.

Usage

```
genCDFInv_quantile(data, type = 1)
```

Arguments

`data`	A numeric vector. The dataset for which the inverse CDF is to be created.
`type`	Integer. Specifies the algorithm used to compute quantiles. See `quantile` for details. Default is `1`.

Details

The function works by wrapping the `quantile` function. The `type` parameter controls the quantile computation method. For example:

- Type 1 corresponds to inverse of the empirical distribution function (default).
- Other types correspond to different quantile algorithms as documented in `quantile`.

Value

A function that takes a single argument, `p`, a numeric vector of probabilities in `[0, 1]`, and returns the corresponding quantiles from the data.

See Also

`quantile`

Examples

```
# Example usage:
data <- c(1, 2, 3, 4, 5)
inv_cdf <- genCDFInv_quantile(data, type = 1)
inv_cdf(c(0.25, 0.5, 0.75))  # Compute the 25th, 50th, and 75th percentiles
```

`generate_gaussian_copula_samples` *Generate Gaussian Copula Samples*

Description

This function generates samples from a Gaussian copula given a specified correlation matrix. The samples are uniformly distributed in `[0, 1]` across dimensions.

Usage

```
generate_gaussian_copula_samples(n, d, rho_matrix)
```

Arguments

`n`	Integer. The number of samples to generate.
`d`	Integer. The dimensionality of the copula.
`rho_matrix`	A `d` x `d` positive-definite correlation matrix.

Details

The function works as follows:

1. Generates multivariate normal samples with the given correlation matrix.
2. Transforms the samples to the uniform distribution `[0, 1]` using the cumulative distribution function (CDF) of the standard normal.

Value

A matrix of size `n` x `d`, where each row represents a sample and each column corresponds to a dimension. The values are uniformly distributed in `[0, 1]`.

Examples

```
# Example usage:
library(MASS)  # Load package for `mvrnorm`
rho_matrix <- matrix(c(1, 0.5, 0.5, 1), nrow = 2)  # 2x2 correlation matrix
samples <- generate_gaussian_copula_samples(n = 1000, d = 2, rho_matrix = rho_matrix)
head(samples)
```

`generate_t_copula_samples` *Generate t-Copula Samples*

Description

This function generates samples from a t-copula given a specified correlation matrix and degrees of freedom. The samples are uniformly distributed in `[0, 1]`. across dimensions.

Usage

```
generate_t_copula_samples(n, d, rho_matrix, df)
```

Arguments

`n`	Integer. The number of samples to generate.
`d`	Integer. The dimensionality of the copula.
`rho_matrix`	A `d` x `d` positive-definite correlation matrix.
`df`	Numeric. The degrees of freedom of the t-distribution. Must be positive.

Details

The function works as follows:

1. Generates multivariate t-distributed samples using the specified correlation matrix and degrees of freedom.
2. Transforms the samples to the uniform distribution `[0, 1]` using the cumulative distribution function (CDF) of the t-distribution.

Value

A matrix of size `n` x `d`, where each row represents a sample and each column corresponds to a dimension. The values are uniformly distributed in `[0, 1]`.

Examples

```
# Example usage:
library(mvtnorm)  # Load package for `rmvt`
rho_matrix <- diag(3)  # 3x3 identity matrix as correlation matrix
samples <- generate_t_copula_samples(n = 1000, d = 3, rho_matrix = rho_matrix, df = 5)
head(samples)
```

gen_number_1 *Generate a New Number for Stepwise Modification*

Description

This function modifies a given contingency table by swapping values between two cells in a stepwise manner, where the change is fixed at a `delta` value of 1. The function randomly selects two cells from the table and adjusts their values by subtracting and adding the `delta` value.

Usage

```
gen_number_1(x)
```

Arguments

x	A contingency table (numeric matrix or table).

Details

This function performs the following steps:

1. Randomly selects two rows and two columns from the table.
2. Ensures that the selected cells have non-zero values.
3. Adjusts the values of the selected cells by subtracting 1 from two cells and adding 1 to the other two.
4. Returns the modified table with stepwise adjustments.

Value

A modified contingency table with stepwise adjustments.

Examples

```
# Example usage with a contingency table:
pair_table <- table(c(1, 2, 2, 3), c(1, 1, 2, 2))
gen_number_1(pair_table)
```

gen_number_max *Generate a New Number for Maximizing Mutual Information*

Description

This function modifies a given contingency table by swapping values between two cells to maximize the mutual information. The function randomly selects two cells from the table and adjusts their values in a way that increases mutual information. The function then returns the modified table with the highest mutual information.

Usage

```
gen_number_max(x)
```

Arguments

x	A contingency table (numeric matrix or table).

Details

This function performs the following steps:

1. Randomly selects two rows and two columns from the table.
2. Checks if the selected cells have non-zero values.
3. Adjusts the values of the selected cells in two different modified tables, `table1` and `table2`.
4. Calculates the mutual information for both modified tables.
5. Returns the table with the higher mutual information.

Value

A modified contingency table with maximized mutual information.

Examples

```
# Example usage with a contingency table:
pair_table <- table(c(1, 2, 2, 3), c(1, 1, 2, 2))
gen_number_max(pair_table)
```

gen_number_min *Generate a New Number for Minimizing Mutual Information*

Description

This function modifies a given contingency table by swapping values between two cells in a way that minimizes the mutual information. The function randomly selects two cells from the table and adjusts their values to reduce mutual information, returning the modified table.

Usage

```
gen_number_min(x)
```

Arguments

x	A contingency table (numeric matrix or table).

Details

This function performs the following steps:

1. Randomly selects two rows and two columns from the table.
2. Ensures that the selected cells have non-zero values.
3. Adjusts the values of the selected cells in the table to minimize mutual information.
4. Returns the modified table with minimized mutual information.

Value

A modified contingency table with minimized mutual information.

Examples

```
# Example usage with a contingency table:
pair_table <- table(c(1, 2, 2, 3), c(1, 1, 2, 2))
gen_number_min(pair_table)
```

`get_mutual_information` *Calculate Mutual Information*

Description

This function calculates the mutual information between two variables based on their joint distribution. Mutual information measures the amount of information obtained about one variable through the other.

Usage

```
get_mutual_information(table)
```

Arguments

`table`	A numeric matrix or table. A contingency table or frequency table of two variables.

Details

The mutual information is calculated using the formula: [I(X, Y) = H(X) + H(Y) - H(X, Y)] where:

- \(H(X) \) is the entropy of variable X,
- \(H(Y) \) is the entropy of variable Y, and
- \(H(X, Y) \) is the joint entropy of X and Y.

Value

A numeric value representing the mutual information between the two variables.

Examples

```
# Example usage with a simple contingency table:
pair_table <- table(c(1, 2, 2, 3), c(1, 1, 2, 2))
get_mutual_information(pair_table)
```

`get_optimal_grid` *Get Optimal Grid Assignment*

Description

This function computes an optimal grid assignment between two variables `x` and `y` based on a third variable `z`. It uses the quantiles of `x` and `y` to segment the data based on the distribution of `z`. Then, it computes the cost of assigning points from `x` to `y` by calculating the counts of `y` values within quantile ranges of `x` and `y`, and then solves the assignment problem using the Hungarian algorithm.

Usage

```
get_optimal_grid(x, y, z)
```

Arguments

`x`	A numeric vector of values representing the first variable.
`y`	A numeric vector of values representing the second variable.
`z`	A numeric vector representing the distribution of the third variable, used to define quantiles for `x` and `y`.

Value

A numeric vector of optimal indices that represents the optimal assignment of `x` values to `y` values.

Examples

```
# Test Case 1: Simple uniform data
x <- rnorm(1000)
y <- rnorm(1000)
z <- sample(1:5, 1000, replace = TRUE)
optimal_assignment <- get_optimal_grid(x, y, z)

# Test Case 2: Data with a skewed distribution
x <- rexp(1000, rate = 1)
y <- rpois(1000, lambda = 2)
z <- sample(1:3, 1000, replace = TRUE)
optimal_assignment <- get_optimal_grid(x, y, z)
```

get_simpsons_paradox_c *Simpson's Paradox Transformation with Copula and Simulated Annealing*

Description

This function simulates the Simpson's Paradox phenomenon by transforming data using Gaussian copulas, optimizing the transformation with simulated annealing, and comparing the results.

Usage

```
get_simpsons_paradox_c(
  x,
  y,
  z,
  corr_vector,
  inv_cdf_type = "quantile_7",
  sd_x = 0.05,
  sd_y = 0.05,
  lambda1 = 1,
  lambda2 = 1,
  lambda3 = 1,
  lambda4 = 1,
  max_iter = 1000,
  initial_temp = 1,
  cooling_rate = 0.99,
  order_vec = NA,
  degree = 5
)
```

Arguments

x	A numeric vector of data points for variable X.
y	A numeric vector of data points for variable Y.
z	A categorical variable representing groups (e.g., factor or character vector).
corr_vector	A vector of correlations for each category of z.
inv_cdf_type	Type of inverse CDF transformation ("quantile_1", "quantile_4", "quantile_7", "quantile_8", "linear", "akima", "poly"). Default is "quantile_7".
sd_x	Standard deviation for perturbations on X (default is 0.05).
sd_y	Standard deviation for perturbations on Y (default is 0.05).
lambda1	Regularization parameter for simulated annealing (default is 1).
lambda2	Regularization parameter for simulated annealing (default is 1).
lambda3	Regularization parameter for simulated annealing (default is 1).

lambda4	Regularization parameter for simulated annealing (default is 1).
max_iter	Maximum iterations for simulated annealing (default is 1000).
initial_temp	Initial temperature for simulated annealing (default is 1.0).
cooling_rate	Cooling rate for simulated annealing (default is 0.99).
order_vec	Manual ordering of grids (default is NA, calculated automatically if not specified).
degree	Degree of polynomial used for polynomial inverse CDF (default is 5).

Value

A list containing:

df_all	The final dataset with original, transformed, and annealed data.
df_res	A simplified version with only the optimized data.

Examples

```
set.seed(123)
n <- 300
z <- sample(c("A", "B", "C"), prob = c(0.3, 0.4, 0.3), size = n, replace = TRUE)
x <- rnorm(n, 10, sd = 5) + 5 * rbeta(n, 5, 3)
y <- 2 * x + rnorm(n, 5, sd = 4)
t <- c(-0.8, 0.8, -0.8)
res <- get_simpsons_paradox_c(x, y, z, t, sd_x = 0.07, sd_y = 0.07, lambda4 = 5)
```

---

get_simpsons_paradox_d

Introduce Simpson's Paradox in Discrete Data

Description

This function modifies contingency tables associated with different levels of a categorical variable to create or highlight Simpson's Paradox using simulated annealing. The paradox occurs when aggregated data trends differ from subgroup trends.

Usage

```
get_simpsons_paradox_d(
  x,
  y,
  z,
  manual_vec,
  target_overall,
  margin,
  margin_overall,
```

```
  max_n = 1000,
  temp = 10,
  log_odds_general = log_odds_dc
)
```

Arguments

x	A vector of categorical values for the first variable.
y	A vector of categorical values for the second variable.
z	A vector indicating levels of a third variable that segments the data.
manual_vec	A numeric vector specifying target log-odds trends for each level of z.
target_overall	A numeric value representing the target log-odds for the aggregated data.
margin	A numeric value for allowed deviation in log-odds within each subgroup.
margin_overall	A numeric value for allowed deviation in aggregated log-odds.
max_n	An integer specifying the maximum number of iterations for the annealing process.
temp	A numeric value for the initial temperature in the annealing process.
log_odds_general	A function to compute the log-odds for a given contingency table (default: log_odds_dc).

Details

This function works by iteratively modifying individual matrices (contingency tables) corresponding to levels of z while respecting log-odds constraints. The overall log-odds of the aggregated table are also adjusted to achieve the specified target_overall. Simulated annealing ensures that the modifications balance between achieving the targets and avoiding overfitting.

Value

A list containing:

- final_df: A data frame representing the modified dataset.
- final_table: A list of modified contingency tables.
- history: A data frame tracking the overall log-odds over iterations.

Examples

```
# Example with predefined contingency tables
set.seed(42)
matrices <- list(
  ta = matrix(c(512, 89, 313, 19), ncol = 2, byrow = TRUE),
  tb = matrix(c(353, 17, 207, 8), ncol = 2, byrow = TRUE),
  tc = matrix(c(120, 202, 205, 391), ncol = 2, byrow = TRUE)
)
df_list <- lapply(seq_along(matrices), function(i) {
  mat <- matrices[[i]]
```

```
  z_level <- names(matrices)[i]
  df <- as.data.frame(as.table(mat))
  colnames(df) <- c("x", "y", "Freq")
  df$z <- z_level
  return(df)
})
final_df <- do.call(rbind, df_list)
expanded_df <- final_df[rep(1:nrow(final_df), final_df$Freq), c("x", "y", "z")]
result <- get_simpsons_paradox_d(
  expanded_df$x, expanded_df$y, expanded_df$z,
  manual_vec = c(-1, -1, -1),
  target_overall = +1,
  margin = 0.2, margin_overall = 0.2, max_n = 200
)
table(expanded_df$x) - table(result$final_df$x)
```

`get_target_corr` *Generate Samples with Target Kendall's Tau Correlation Using a Copula Approach*

Description

This function generates two variables with a specified target Kendall's tau correlation using copula-based methods. The user can specify the type of copula (Gaussian or t), the type of inverse CDF method to apply to the variables, and the degree of the polynomial interpolation if applicable.

Usage

```
get_target_corr(
  x1,
  x2,
  target_corr_kendall,
  copula_type = "gaussian",
  inv_cdf_type = "quantile_7",
  degree = 10
)
```

Arguments

`x1`	A numeric vector. The first dataset used for generating inverse CDFs.
`x2`	A numeric vector. The second dataset used for generating inverse CDFs.
`target_corr_kendall`	A numeric value. The desired target Kendall's tau correlation between the two generated variables.
`copula_type`	A string. The type of copula to use, either "gaussian" or "t" (default is "gaussian").

inv_cdf_type	A string. The type of inverse CDF method to use. Options include: "quantile_1", "quantile_4", "quantile_7", "quantile_8", "linear", "akima", "poly" (default is "quantile_7").
degree	An integer. The degree of the polynomial interpolation (default is 10).

Details

This function works by:

1. Generating two variables using the specified copula type (Gaussian or t) with the target Kendall's tau correlation.
2. Applying the chosen inverse CDF transformation to the generated copula samples.
3. Returning the modified variables that have the target correlation.

Value

A list containing two components: x1 and x2, which are the modified versions of the input datasets x1 and x2 with the desired target Kendall's tau correlation.

See Also

gaussian_copula_two_vars, t_copula_two_vars, genCDFInv_quantile, genCDFInv_linear, genCDFInv_akima, genCDFInv_poly

Examples

```
# Example usage:
x1 <- ChickWeight$weight
x2 <- ChickWeight$Time
cor(x1, x2, method = "kendall")  # Calculate original Kendall's tau correlation
res <- get_target_corr(x1, x2, target_corr_kendall = 0,
                       copula_type = "gaussian", inv_cdf_type = "poly")
cor(res$x1, res$x2, method = "kendall")  # Calculate modified Kendall's tau correlation
```

---

get_target_entropy *Get Target Entropy*

Description

This function adjusts the mutual information between two categorical variables (x and y) by modifying their contingency table using simulated annealing to reach a target entropy. The function first calculates the range of possible entropy values (min and max) and checks if the target entropy lies within that range. If so, it adjusts the mutual information to reach the target entropy, either by increasing or decreasing it, depending on the initial entropy value.

Usage

```
get_target_entropy(x, y, target_entropy, max_n = 10000, epsilon = 0.001)
```

Arguments

x	A vector of categorical values representing variable x.
y	A vector of categorical values representing variable y.
target_entropy	The target entropy value to be reached.
max_n	Maximum number of iterations for the optimization process (default is 10,000).
epsilon	The tolerance value for determining if the target entropy has been reached (default is 0.001).

Value

A list containing:

- final_df: A dataframe with the adjusted contingency table.
- final_table: The final contingency table after adjustments.
- history: The history of the optimization process.
- max_mut: The maximum mutual information found.
- min_mut: The minimum mutual information found.

Examples

```
set.seed(33)
df <- data.frame(
  x = sample(paste("Categ", 1:4), 10000, replace = TRUE),
  y = sample(paste("Categ", 10:4), 10000, replace = TRUE)
)
target_entropy <- 1  # Set your target entropy here
res <- get_target_entropy(df$x, df$y, target_entropy)
```

---

log_odds_dc *Log-Odds Calculation for Concordant and Discordant Pairs*

Description

This function calculates the log-odds ratio for concordant and discordant pairs based on the contingency table provided. The log-odds ratio is defined as the natural logarithm of the ratio of concordant to discordant pairs.

Usage

```
log_odds_dc(tab)
```

Arguments

tab	A contingency table (matrix or data frame) containing counts of pairs for each combination of outcomes.

Value

The log-odds ratio calculated as the natural logarithm of the ratio of concordant pairs to discordant pairs. If discordant pairs are zero, it returns `Inf` to avoid division by zero.

Examples

```
# Example contingency table
tab <- matrix(c(10, 5, 7, 8), nrow = 2)
log_odds_dc(tab)
```

---

objective_function_SL *Objective Function for Structural Learning (SL)*

Description

This function calculates the objective function for a structural learning task. It computes multiple components such as the total variation (TV) distance between original and generated datasets (`X` vs. `X_prime`, `Y` vs. `Y_prime`), the changes in regression coefficients (`beta0` and `beta1`), the R^2 differences for each category in `Z`, and the inter-cluster centroid distances. The loss function combines these components using penalty parameters (`lambda1`, `lambda2`, `lambda3`, `lambda4`).

Usage

```
objective_function_SL(
  X_prime,
  Y_prime,
  X,
  Y,
  Z,
  p,
  beta0_orig,
  beta1_orig,
  lambda1,
  lambda2,
  lambda3,
  lambda4,
  R2_orig,
  printc = FALSE
)
```

Arguments

`X_prime`	A numeric vector representing the generated values of `X`.
`Y_prime`	A numeric vector representing the generated values of `Y`.
`X`	A numeric vector representing the original values of `X`.

Y	A numeric vector representing the original values of Y.
Z	A categorical vector representing the categories for each observation.
p	A numeric vector representing the target correlation values for each category in Z.
beta0_orig	The original intercept value for the regression model.
beta1_orig	The original slope value for the regression model.
lambda1	Penalty parameters to control the importance of different loss components.
lambda2	Penalty parameters to control the importance of different loss components.
lambda3	Penalty parameters to control the importance of different loss components.
lambda4	Penalty parameters to control the importance of different loss components.
R2_orig	The original R^2 value for the model (not used directly in the calculation but might be for reference).
printc	A boolean flag to control printing of intermediate values for debugging.

Value

A numeric value representing the total loss calculated by the objective function.

Examples

```
# Test Case 1: Simple random data with normal distribution
set.seed(123)
X <- rnorm(100)
Y <- rnorm(100)
Z <- sample(1:3, 100, replace = TRUE)
X_prime <- rnorm(100)
Y_prime <- rnorm(100)
p <- c(0.5, 0.7, 0.9)
beta0_orig <- 0
beta1_orig <- 1
lambda1 <- lambda2 <- lambda3 <- lambda4 <- 1
R2_orig <- 0.9
loss <- objective_function_SL(X, Y, Z, X_prime, Y_prime, p, beta0_orig, beta1_orig,
                              lambda1, lambda2, lambda3, lambda4, R2_orig)
print(loss)

# Test Case 2: Skewed data with different categories and a larger lambda for penalty
X <- rexp(100)
Y <- rpois(100, lambda = 2)
Z <- sample(1:4, 100, replace = TRUE)
X_prime <- rnorm(100)
Y_prime <- rnorm(100)
p <- c(0.3, 0.5, 0.8, 0.6)
beta0_orig <- 0.5
beta1_orig <- 1.5
lambda1 <- lambda2 <- lambda3 <- 0.5
lambda4 <- 2
R2_orig <- 0.85
```

```
loss <- objective_function_SL(X, Y, Z, X_prime, Y_prime, p, beta0_orig, beta1_orig,
                              lambda1, lambda2, lambda3, lambda4, R2_orig)
print(loss)
```

`plot_log_odds` *Plot Log-Odds Before and After Transformation*

Description

This function calculates the log-odds ratio before and after applying a transformation to multiple matrices, and generates a bar plot comparing the log-odds values. The log-odds are calculated using a specified function (default is `log_odds_dc`).

Usage

```
plot_log_odds(
  matrices,
  new_matrices,
  names_matrices,
  log_odds_general = log_odds_dc
)
```

Arguments

`matrices`	A list of matrices for which the log-odds are calculated before the transformation.
`new_matrices`	A list of matrices for which the log-odds are calculated after the transformation.
`names_matrices`	A vector of names corresponding to the matrices in `matrices` and `new_matrices`.
`log_odds_general`	A function used to calculate the log-odds (default is `log_odds_dc`).

Value

A bar plot showing the log-odds before and after the transformation for each matrix and overall.

Examples

```
# Example matrices and names
matrices <- list(matrix(c(1, 2, 3, 4), nrow = 2), matrix(c(2, 3, 4, 5), nrow = 2))
new_matrices <- list(matrix(c(5, 6, 7, 8), nrow = 2), matrix(c(4, 5, 6, 7), nrow = 2))
names_matrices <- c("Matrix1", "Matrix2")
plot_log_odds(matrices, new_matrices, names_matrices)
```

`simulated_annealing_MI` *Simulated Annealing Algorithm with Target Entropy Stopping Condition*

Description

This function performs simulated annealing to optimize a given objective (entropy, mutual information, etc.) using a given table modification function. The optimization stops once the target entropy is reached or after a maximum number of iterations.

Usage

```
simulated_annealing_MI(
  initial_table,
  obj,
  gen_fn,
  target,
  max_n = 5000,
  temp = 10,
  maxim = TRUE,
  readj = FALSE
)
```

Arguments

`initial_table`	A contingency table to start the optimization.
`obj`	An objective function that calculates the value to be optimized (e.g., entropy, mutual information).
`gen_fn`	A function that generates a new table based on the current table.
`target`	The target value for the objective function (e.g., target entropy).
`max_n`	The maximum number of iterations to run the algorithm (default is 5000).
`temp`	The initial temperature for the simulated annealing process (default is 10).
`maxim`	Logical: Should the algorithm maximize (TRUE) or minimize (FALSE) the objective function (default is TRUE).
`readj`	Logical: If TRUE, the algorithm is in a readjusting state (default is FALSE).

Value

A list containing:

- `best`: The best table found during the optimization process.
- `best_eval`: The best evaluation value (objective function value).
- `n`: The number of iterations completed.
- `mutual_info_history`: A data frame with the history of mutual information values (or objective function values) during each iteration.

Examples

```
# Example of using simulated annealing for entropy maximization:
initial_table <- matrix(c(5, 3, 4, 2), nrow = 2, ncol = 2)
obj <- entropy_pair  # Example entropy function
gen_fn <- gen_number_max  # Example generation function
target <- 0.5
result <- simulated_annealing_MI(initial_table, obj, gen_fn, target)
```

---

simulated_annealing_SL

Simulated Annealing Optimization with Categorical Variable and R^2 Differences

---

Description

This function implements the Simulated Annealing algorithm to optimize a solution based on the total variation distance, changes in regression coefficients, R-squared differences, and inter-cluster distance, with respect to a set of categorical and continuous variables.

Usage

```
simulated_annealing_SL(
  X,
  Y,
  Z,
  X_st,
  Y_st,
  p,
  sd_x = 0.05,
  sd_y = 0.05,
  lambda1 = 1,
  lambda2 = 1,
  lambda3 = 1,
  lambda4 = 1,
  max_iter = 1000,
  initial_temp = 1,
  cooling_rate = 0.99
)
```

Arguments

X	A numeric vector or matrix of input data (independent variable).
Y	A numeric vector of the dependent variable (target).
Z	A categorical variable (vector), used for grouping data in the analysis.
X_st	A numeric vector of starting values for the composition method of X.

Y_st	A numeric vector of starting values for the composition method of Y.
p	A numeric vector representing the target R^2 values for each category in Z.
sd_x	Standard deviation for the noise added to X during the perturbation (default is 0.05).
sd_y	Standard deviation for the noise added to Y during the perturbation (default is 0.05).
lambda1	Regularization parameter for the total variation distance term (default is 1).
lambda2	Regularization parameter for the coefficient difference term (default is 1).
lambda3	Regularization parameter for the R^2 difference term (default is 1).
lambda4	Regularization parameter for the inter-cluster distance term (default is 1).
max_iter	Maximum number of iterations for the annealing process (default is 1000).
initial_temp	Initial temperature for the annealing process (default is 1.0).
cooling_rate	The rate at which the temperature cools down during annealing (default is 0.99).

Value

A list with the optimized values of X_prime and Y_prime.

---

sinkhorn_algorithm *Sinkhorn Algorithm for Matrix Scaling*

Description

This function applies the Sinkhorn-Knopp algorithm to adjust the row and column sums of a matrix to match the target sums. The algorithm iteratively scales the rows and columns by updating scaling factors (alpha and beta) until convergence or the maximum number of iterations is reached.

Usage

```
sinkhorn_algorithm(initial_table, obj, max_iter = 500, tolerance = 1e-05)
```

Arguments

initial_table	A matrix to be adjusted using the Sinkhorn algorithm.
obj	An objective function to evaluate the matrix (e.g., entropy, mutual information).
max_iter	The maximum number of iterations for the algorithm (default is 500).
tolerance	The convergence tolerance. If the change in the objective function is smaller than this, the algorithm stops (default is 1e-5).

Value

A list containing:

- `updated_table`: The matrix after Sinkhorn scaling.
- `new_mut`: The objective function value for the scaled matrix.
- `iter`: The number of iterations performed.
- `mutual_info_history`: A data frame with the history of objective function values during each iteration.

Examples

```
initial_table <- matrix(c(5, 3, 4, 2), nrow = 2, ncol = 2)
obj <- entropy_pair  # Example entropy function
result <- sinkhorn_algorithm(initial_table, obj)
```

---

softmax *Softmax Function with Special Handling for Infinite Values*

Description

This function computes the softmax of a vector `x`, with special handling for infinite values. The softmax function transforms input values into a probability distribution by exponentiating each value, then normalizing by the sum of all exponentiated values. The function ensures numerical stability, particularly when dealing with very large or very small values, and handles cases where the values are infinite (`Inf`).

Usage

```
softmax(x)
```

Arguments

`x`	A numeric vector for which the softmax function will be calculated.

Value

A numeric vector of the same length as `x`, where the values represent probabilities summing to 1.

Examples

```
softmax(c(10, 5, 2))
softmax(c(Inf, -Inf, 0))
softmax(c(-Inf, -Inf, -Inf))
```

`t_copula_two_vars` *Generate t-Copula Samples for Two Variables*

Description

This function generates samples from a t-copula with two variables, given the specified correlation coefficient between the variables.

Usage

```
t_copula_two_vars(n, p)
```

Arguments

`n`	Integer. The number of samples to generate.
`p`	Numeric. The correlation coefficient (ρ) between the two variables. Must be in the range `[-1, 1]`.

Details

The function internally constructs a correlation matrix for two variables:

$$\rho_matrix = \begin{bmatrix} 1 & p \\ p & 1 \end{bmatrix}$$

It then calls `generate_t_copula_samples` to generate samples using a t-distribution with 5 degrees of freedom.

Value

A matrix of size `n x 2`, where each row represents a sample, and each column corresponds to one of the two variables. The values are uniformly distributed in `[0, 1]`.

See Also

`generate_t_copula_samples`

Examples

```
# Example usage:
samples <- t_copula_two_vars(n = 1000, p = 0.7)
head(samples)
```

Index

approxfun, *7*
aspline, *6*
augment_matrix_random_block, 2

calculate_tv_distance_empirical, 3

ecdf, *6–8*
entropy_pair, 4

gaussian_copula_two_vars, 5, *21*
gen_number_1, 12
gen_number_max, 13
gen_number_min, 14
genCDFInv_akima, 6, *21*
genCDFInv_linear, 7, *21*
genCDFInv_poly, 8, *21*
genCDFInv_quantile, 9, *21*
generate_gaussian_copula_samples, *5*, 10
generate_t_copula_samples, 11, *30*
get_mutual_information, 15
get_optimal_grid, 16
get_simpsons_paradox_c, 17
get_simpsons_paradox_d, 18
get_target_corr, 20
get_target_entropy, 21

lm, *8*
log_odds_dc, 22

objective_function_SL, 23

plot_log_odds, 25
poly, *8*

quantile, *9*

simulated_annealing_MI, 26
simulated_annealing_SL, 27
sinkhorn_algorithm, 28
softmax, 29

t_copula_two_vars, *21*, 30