Package: rarefun (via r-universe)

June 3, 2026

Type Package

Title Functions for Rare Events Analysis

Version 0.1.0

Description Functions for detecting and analyzing rare events in data. Implements isolation forest (Liu et al., 2008, <doi:10.1109/ICDM.2008.17>) and clustering for anomaly detection in time series residuals. Decomposes time series using LOESS (Locally Estimated Scatterplot Smoothing) or STL (Seasonal-Trend decomposition using LOESS). Detects marine heatwaves and cold spells following Hobday et al. (2016) <doi:10.1016/j.pocean.2015.12.014>. Provides goodness-of-fit tests for quantile regression (Haupt et al., 2011, <doi:10.1080/02664763.2011.573542>), partial dependence with quantile random forests, MCC (Matthews Correlation Coefficient) computation and testing, knee-point detection via the Kneedle algorithm (Satopaa et al., 2011, <doi:10.1109/ICDCSW.2011.20>), and spatial point matching.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.1.0)

Imports RANN, boot, dbscan, geosphere, isotree, Rdpack, parallel, pdp, stats

Suggests testthat (>= 3.0.0), dplyr, ggplot2, heatwaveR, patchwork, quantmod, quantreg, ranger, viridis

URL https://github.com/vlyubchich/rarefun

BugReports https://github.com/vlyubchich/rarefun/issues

RdMacros Rdpack

LazyData true

Config/roxygen2/version 8.0.0

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

Date/Publication 2026-05-29 15:27:38 UTC

RemoteUrl https://github.com/vlyubchich/rarefun

RemoteRef HEAD

RemoteSha 62c88016a40f1210f4c25ae23ade7bd7448c50e2

Contents

`Annapolis` *Annapolis Daily Daymet Time Series (1980-2024)*

Description

Daily maximum temperature and precipitation for a single Daymet pixel near Annapolis, Maryland (USA). These data were retrieved via the package `daymetr` version 1.7.1 and pared down for compact examples in this package.

- Source: Daymet (ORNL DAAC / NASA ESDIS) via the package `daymetr`.
- Units: `tmax` in degrees Celsius; `pcp` in millimeters per day.
- Coverage: `1980-01-01` through `2024-12-31` (inclusive; 365-day years excluding 31 December from leap years), one location (Annapolis, MD; approx. 38.9784° N, 76.4922° W).

Usage

```
Annapolis
```

Format

A tibble (data frame) with the following columns:

- `date` Date. Calendar date.
- `tmax` numeric. Daily maximum 2 m air temperature (°C).
- `pcp` numeric. Daily total precipitation (mm/day).

Source

Thornton, M. M., Shrestha, R., Wei, Y., Thornton, P. E., & Kao, S.-C. (2022). Daymet: Daily Surface Weather Data on a 1-km Grid for North America, Version 4 R1 (Version 4.1). ORNL Distributed Active Archive Center. doi:10.3334/ORNLDAAC/2129 Date Accessed: 2025-11-18.

Examples

```
data(Annapolis)
head(Annapolis)

# Quick check of data
summary(Annapolis)

# Minimal plot (if graphics device available)
plot(Annapolis$date, Annapolis$tmax, type = "l",
     xlab = "Date", ylab = "Tmax (°C)",
     las = 1)
```

`gof_qr` *Goodness-of-fit Measures for Quantile Regression*

Description

Calculate goodness-of-fit measures for quantile regression as given in Haupt et al. (2011). Specifically, for a given quantile v, let $p_v(u) = (v - I_{u<0})u$, where I is an indicator function with outputs $\{0, 1\}$; and let y_i be the observed values, $\hat{y}_i(v)$ fitted values for the sample of size n ($i = 1, \ldots, n$), and y_v be the observed v-quantile. Then, R^1–an analogue of R^2–is defined as

$$R^1(v) = 1 - \frac{\sum_{i=1}^{n} p_v(y_i - \hat{y}_i(v))}{\sum_{i=1}^{n} p_v(y_i - y_v)},$$

and average v-weighted absolute error (ATWE) is

$$ATWE(v) = n^{-1} \sum_{i=1}^{n} p_v(y_i - \hat{y}_i(v)).$$

Higher R^1 and lower ATWE are preferred.

Usage

```
gof_qr(obs, pred, quantiles = NULL)
```

Arguments

`obs`	A numeric vector or a column matrix of observed values.
`pred`	A numeric vector or a matrix of predicted quantiles. Columns represent different quantiles.
`quantiles`	Numeric vector, the quantiles for which the predictions were made (default: NULL). If `pred` is a vector, `length(quantiles)` must be 1; if `pred` is a matrix, `length(quantiles)` must be `ncol(pred)`. If NULL, the quantiles are guessed from `colnames(pred)` assuming `pred` are predictions from `ranger` or `rq`.

Details

This function computes two goodness-of-fit measures for quantile regression models. R1 is analogous to the coefficient of determination (R^2) used in ordinary least squares regression, while ATWE measures the average prediction error weighted by the quantile. Both measures are computed using the check function $p_v(u)$ which asymmetrically penalizes over- and under-prediction based on the target quantile v.

Value

A matrix with rows for R1 and ATWE, with one column per quantile (matching the number of columns in pred).

References

Haupt H, Kagerer K, Schnurbus J (2011). "Cross-validating fit and predictive accuracy of nonlinear quantile regressions." *Journal of Applied Statistics*, **38**(12), 2939–2954. doi:10.1080/02664763.2011.573542.

See Also

partial_qrf for partial dependence with quantile random forests.

Examples

```
# Example: Quantile regression goodness-of-fit on swiss data
# Select 30% of data for testing
n <- nrow(swiss)
testindex <- sample(1:n, 0.3 * n, replace = FALSE)
# Desired quantiles
qs <- c(0.025, 0.5, 0.975)

# Example 1: Quantile random forest
# Fit a model on the training set
qrf <- ranger::ranger(Examination ~ ., data = swiss[-testindex, ], quantreg = TRUE)
# Predict on the testing set
pred_qrf <- predict(qrf, swiss[testindex,], type = "quantiles", quantiles = qs)$predictions
# Get goodness-of-fit summary on the testing set
gof_qr(swiss[testindex, "Examination"], pred_qrf)

# Example 2: Quantile regression
# Fit a model on the training set
qrm <- quantreg::rq(Examination ~ ., data = swiss[-testindex, ], tau = qs)
# Predict on the testing set
pred_qrm <- predict(qrm, newdata = swiss[testindex, ])
# Get goodness-of-fit summary on the testing set
gof_qr(swiss[testindex, "Examination"], pred_qrm)
```

`kneedle` *Detect Knee Point in a Curve Using the Kneedle Algorithm*

Description

This function implements the Kneedle algorithm to detect the "knee" or "elbow" point in a curve, as described by Satopaa et al. (2011). The knee point is the point of maximum curvature, often used to identify a transition in data behavior (e.g., optimal clustering parameters). The algorithm normalizes the input data, computes the difference between the curve and a reference line, and identifies the knee based on peaks in the difference curve, adjusted by a sensitivity parameter.

Usage

```
kneedle(x, y, decreasing = NULL, concave = NULL, sensitivity = 1)
```

Arguments

`x`	A numeric vector of x coordinates, strictly increasing or decreasing.
`y`	A numeric vector of y coordinates, same length as `x`.
`decreasing`	Logical, indicating if the curve is decreasing (TRUE) or increasing (FALSE). If NULL, the function estimates the direction based on the difference between the first and last y values (default: NULL).
`concave`	Logical, indicating if the curve is concave (TRUE) or convex (FALSE). If NULL, the function estimates concavity based on the average second derivative of the data (default: NULL).
`sensitivity`	Numeric, sensitivity for detecting the knee point. Higher values make the algorithm more selective, requiring a stronger peak to identify the knee (default: 1).

Details

The code is adapted from https://github.com/etam4260/kneedle/blob/main/R/kneedle.R (copyright 2022 kneedle authors), licensed under the MIT license.

The Kneedle algorithm detects the knee point by normalizing the input data to [0, 1], computing the difference between the normalized curve and a reference line (x = y or x = 1 - y, depending on direction and concavity), and identifying peaks in the difference curve. The first peak exceeding a threshold (adjusted by `sensitivity`) is selected as the knee. The function automatically estimates `decreasing` and `concave` if not specified, using the slope from the first to last point and the average second derivative, respectively.

Value

A numeric vector of length 2 containing the x and y coordinates of the detected knee point. If no knee is found, returns `c(NA, NA)`. This can occur when the curve is too smooth or lacks a clear inflection point.

References

Satopaa V, Albrecht J, Irwin D, Raghavan B (2011). “Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior.” In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, 166–171. doi:10.1109/ICDCSW.2011.20.

Examples

```
# Example with synthetic data
x <- 1:10
y <- c(1, 2, 3, 4, 10, 20, 30, 40, 50, 60)
knee <- kneedle(x, y, sensitivity = 1)
plot(x, y, type = "l", main = "Knee Point Detection")
points(knee[1], knee[2], col = "red", pch = 19)
```

---

match_spatial_points *Match Spatial Points between Two Datasets*

Description

Matches each point in the first dataset (`A`) to the nearest point in the second dataset (`B`) based on spatial coordinates (latitude and longitude). The function supports both fast Euclidean distance-based matching (using `RANN::nn2`) and slower, more accurate geodesic distance-based matching (using `geosphere::distGeo`). Geodesic distances are always reported in the output, regardless of the matching method. An optional maximum distance threshold can be applied to filter matches.

Usage

```
match_spatial_points(A, B, max_dist = Inf, fast = TRUE, ...)
```

Arguments

`A`	A data frame containing the first set of points with columns `id` (character or numeric identifier), `lat` (latitude in degrees, -90 to 90), and `lon` (longitude in degrees, -180 to 180).
`B`	A data frame containing the second set of points with columns `id` (character or numeric identifier), `lat` (latitude in degrees, -90 to 90), and `lon` (longitude in degrees, -180 to 180).
`max_dist`	Numeric, the maximum geodesic distance (in meters) for a valid match. Points with distances exceeding this threshold are assigned `NA` in the output. Default is `Inf` (no threshold).
`fast`	Logical, indicating whether to use fast Euclidean distance-based matching (`TRUE`, default) via `RANN::nn2` or slower geodesic distance-based matching (`FALSE`) via `geosphere::distGeo`. Regardless of the method, final distances are computed as geodesic distances.
`...`	Additional arguments passed to `RANN::nn2` (if `fast = TRUE`).

Details

The function matches each point in A to the nearest point in B based on spatial coordinates. If fast = TRUE, it uses Euclidean distances for matching (faster but less accurate for large distances) via RANN::nn2. If fast = FALSE, it computes geodesic distances for matching using geosphere::distGeo, which is more accurate but slower. In both cases, the reported distance_m is the geodesic distance. If max_dist is specified, matches exceeding this distance are replaced with NA. The function handles empty inputs by returning an empty data frame with appropriate column names.

Value

A data frame with one row per point in A, containing the following columns:

- id_A: Identifier of the point from A.
- lat_A: Latitude of the point from A.
- lon_A: Longitude of the point from A.
- id_B: Identifier of the nearest point from B (or NA if no match within max_dist).
- lat_B: Latitude of the nearest point from B (or NA).
- lon_B: Longitude of the nearest point from B (or NA).
- distance_m: Geodesic distance (in meters) to the nearest point in B (or NA if no match).

Column names are dynamically prefixed with the names of the input data frames (e.g., id_Aa if A is named Aa).

See Also

distGeo, nn2

Examples

```
# Example datasets
Aa <- data.frame(
  id = c("fish1", "fish2", "fish3"),
  lat = c(40.7128, 40.7228, 40.7328),
  lon = c(-74.0060, -74.0160, -74.0260)
)
Bb <- data.frame(
  id = c("grid1", "grid2"),
  lat = c(40.7000, 40.7500),
  lon = c(-74.0000, -74.0200)
)

# Fast matching (Euclidean-based)
match_spatial_points(Aa, Bb, fast = TRUE)

# Accurate matching (geodesic-based)
match_spatial_points(Aa, Bb, fast = FALSE)

# Apply maximum distance threshold
match_spatial_points(Aa, Bb, max_dist = 2000)
```

`mcc` *Calculate and Test the Matthews Correlation Coefficient (MCC)*

Description

Based on two vectors of binary values, compute the confusion matrix and the MCC (mean square contingency coefficient). The function implements two methods for testing significance: 1. A parametric chi-square test. 2. A non-parametric bootstrap test for a more robust p-value.

Usage

```
mcc(
  x,
  y,
  positive_class = NULL,
  bootstrap_reps = 999,
  confidence = 0.95,
  ts = FALSE,
  l = 5,
  sim = "fixed",
  ...
)
```

Arguments

`x`	A vector of binary labels. Can be numeric (0/1), logical (TRUE/FALSE), character, or factor.
`y`	A vector of binary labels, of the same type and length as `x`.
`positive_class`	An optional value explicitly specifying the "positive" class label. If `NULL`, the function will infer it.
`bootstrap_reps`	The number of bootstrap replicates for p-value calculation. Default is 999. Set to 0 to disable bootstrapping.
`confidence`	The confidence level for the bootstrap confidence interval (default: 0.95). Must be between 0 and 1 (exclusive).
`ts`	A logical flag indicating if the data are time series. Default is FALSE. If TRUE, a version of bootstrap for time series is applied to account for potential autocorrelation; the data are then assumed to be ordered in time.
`l`	The block length for the time series bootstrap (default: 5). Only used if `ts` is TRUE, see `?boot::tsboot`.
`sim`	The type of simulation for the time series bootstrap (default: "fixed"). Options are "fixed" (moving block bootstrap) or "geom" (stationary bootstrap), see `?boot::tsboot`.
`...`	Additional arguments passed to `boot::tsboot` when `ts` is TRUE.

Details

The MCC is a robust metric for binary classification, especially on imbalanced data. It ranges from -1 (perfect negative correlation) to +1 (perfect positive correlation). For 2x2 confusion matrices:

$$MCC = \frac{TP\,TN - FP\,FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

where: - TP: True Positives - TN: True Negatives - FP: False Positives - FN: False Negatives

and

$$|MCC| = \sqrt{\frac{\chi^2}{n}}$$

The **chi-square test** is fast but assumes that each observation is independent, which is often not true for time series data.

For Time Series: Both standard chi-square and standard bootstrapping assume data independence. If the `ts` parameter is set to TRUE, the function applies a version of bootstrapping suitable for time series data to account for potential autocorrelation. The data are assumed to be ordered in time. The block length `l` can be adjusted based on the expected autocorrelation structure. The chi-square test is still provided for reference but should be interpreted with caution. The bootstrap p-value is more reliable for time series data.

The **bootstrap test** is more computationally intensive but does not assume independence and is more robust, especially for small sample sizes or when the data may be autocorrelated. The bootstrap p-value is calculated as the proportion of bootstrap MCC values that are as extreme or more extreme than the observed MCC, using a two-tailed test. Confidence intervals for the MCC are also provided based on the bootstrap distribution. The function uses the `boot` package for bootstrapping.

Value

A list containing:

- `confusion_matrix`: The 2x2 confusion matrix.
- `mcc`: The Matthews Correlation Coefficient (-1 to +1).
- `chi_square_test`: The output of `chisq.test()`. `$p.value` is the parametric p-value.
- `mcc_bootstrap_pv`: The p-value from the bootstrap permutation test.
- `mcc_bootstrap_ci`: The bootstrap confidence interval for the MCC.
- `mcc_bootstrap_reps`: The number of bootstrap replicates used.
- `positive_class`: The positive class label used.
- `confidence`: The confidence level for the bootstrap confidence interval.
- `ts`: Whether the data were treated as time series.

See Also

`chisq.test`, `tsboot`

Examples

```
# Example 1: A clear, significant correlation
x_vals <- rep(c(1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1), 3)
y_vals <- rep(c(1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0), 3)
mcc_results <- mcc(x_vals, y_vals, positive_class = 1)
print(mcc_results$confusion_matrix)
print(paste("MCC:", round(mcc_results$mcc, 3)))
print(paste("Chi-Square p-value:", round(mcc_results$chi_square_test$p.value, 4)))
print(paste("Bootstrap p-value:", round(mcc_results$mcc_bootstrap_pv, 4)))

# Example 2: No significant correlation
x_rand <- rep(c(1, 0, 1, 0, 1, 0, 1, 0, 1, 0), 3)
y_rand <- rep(c(1, 1, 0, 0, 1, 1, 0, 0, 1, 0), 3)
mcc_rand <- mcc(x_rand, y_rand, bootstrap_reps = 999)
print(paste("MCC:", round(mcc_rand$mcc, 3)))
print(paste("Bootstrap p-value:", round(mcc_rand$mcc_bootstrap_pv, 4)))

# Example 3: Time series data with autocorrelation in both series
n <- 100
set.seed(12345)
ts_x <- rbinom(n, 1, 0.3)
ts_y <- ifelse(runif(n) < 0.3,
               ts_x,
               1 - ts_x)
# Introduce autocorrelation
for (i in 2:n) {
   if (runif(1) < 0.7) {
       ts_x[i] <- ts_x[i - 1]
       ts_y[i] <- ts_y[i - 1]
   }
}

# Check autocorrelation at lag 1
mcc(ts_x, dplyr::lag(ts_x), ts = TRUE, l = 7)$mcc
acf(ts_x, main = "ACF of X Time Series, treated as numeric")
mcc(ts_y, dplyr::lag(ts_y), ts = TRUE, l = 7)$mcc
acf(ts_y, main = "ACF of Y Time Series")

# Visualize the time series
plot.ts(cbind(ts_x, ts_y),
       main = "Time Series of X and Y", xlab = "Time")

# Calculate MCC treating data as time series
mcc_ts <- mcc(ts_x,
              ts_y,
              positive_class = 1,
              ts = TRUE, l = 7, bootstrap_reps = 999)
print(paste("MCC:", round(mcc_ts$mcc, 3)))
print(paste("Chi-Square p-value:", round(mcc_ts$chi_square_test$p.value, 4)))
print(paste("Bootstrap p-value:", round(mcc_ts$mcc_bootstrap_pv, 4)))
print(paste("Bootstrap CI:",
   paste(round(mcc_ts$mcc_bootstrap_ci, 3), collapse = " to ")))
```

partial_qrf *Partial Dependence Plot for Quantile Random Forest*

Description

Get a data frame for partial dependence plots from a quantile random forest. The plots can be obtained using plotting functions from other packages.

Usage

```
partial_qrf(object, pred.var, Q = c(0.05, 0.5, 0.95), ...)
```

Arguments

`object`	a `ranger` quantile random forest object.
`pred.var`	character string giving the names of the predictor variables of interest (see `partial`).
`Q`	a numeric vector of probabilities for which the plot is desired (default: `c(0.05, 0.5, 0.95)`). Values should be between 0 and 1.
`...`	other arguments passed to `partial`.

Value

A `data.frame` with the following columns: the predictor variables supplied in `pred.var`, response variable (the name is extracted from the `ranger` function call), and, if `length(Q) > 1`, one more column named `"Quantile"` (for an appropriate order of labels in the plots, `Quantile` is a factor).

See Also

`partial`, `ranger`

Examples

```
if (requireNamespace("ranger", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)

  # fit a quantile random forest
  qrf <- ranger::ranger(Examination ~ ., data = swiss, quantreg = TRUE, num.trees = 10)

  # 1) a plot for one quantile
  partial_qrf(qrf, pred.var = "Agriculture", Q = 0.5) |>
      ggplot(aes(Agriculture, Examination)) +
          geom_line()

  # 2) a plot for several quantiles, with color scale
  if (requireNamespace("viridis", quietly = TRUE)) {
```

```
  library(viridis)
  partial_qrf(qrf, pred.var = "Agriculture") |>
     ggplot(aes(Agriculture, Examination, group = Quantile, color = Quantile)) +
         geom_line() +
         scale_color_viridis(discrete = TRUE) +
         theme_minimal()
}

# 3) a plot for one quantile for 2 predictors (use small grid.resolution to save time)
df <- partial_qrf(qrf, pred.var = c("Agriculture", "Catholic"), Q = 0.5,
    grid.resolution = 5)
if (requireNamespace("viridis", quietly = TRUE)) {
  ggplot(df, aes(Agriculture, Catholic)) +
      geom_tile(aes(fill = Examination)) +
      scale_fill_viridis() +
      theme_minimal() +
      labs(fill = "Median\nexamination")
}

# 4) a plot for each predictor
varnames <- qrf$forest$independent.variable.names
qs <- c(0.05, 0.5, 0.95)
ddf <- lapply(varnames, function(vn) partial_qrf(qrf, pred.var = vn, Q = qs))
if (requireNamespace("viridis", quietly = TRUE) &&
    requireNamespace("patchwork", quietly = TRUE)) {
  library(viridis)
  library(patchwork)
  yrange <- range(sapply(ddf, function(x) range(x[, 2])))
  plist <- lapply(seq_along(varnames), function(vi) {
      ggplot(ddf[[vi]], aes(x = .data[[varnames[vi]]], y = Examination,
          group = Quantile, color = Quantile)) +
          geom_line() +
          scale_color_viridis(discrete = TRUE) +
          theme_minimal() +
          ylim(yrange[1], yrange[2]) +
          theme(axis.title.y = element_blank())
  })
  wrap_plots(plist) + plot_layout(guides = "collect")
}
}
```

`rare_dbscan` *Detect Rare Events Using DBSCAN*

Description

This function applies the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm of Ester et al. (1996) to identify rare events (anomalies) in a dataset using the `dbscan`

package. Points assigned to the noise cluster (label 0) are considered rare events. The function computes cluster assignments and returns anomaly labels and scores based on the DBSCAN model.

Usage

```
rare_dbscan(x, eps = NULL, minPts = NULL, scale = TRUE, ...)
```

Arguments

`x`	A numeric matrix or data frame with no missing values. Rows are observations, and columns are features.
`eps`	Numeric, the maximum distance between two points for them to be considered in the same neighborhood (default: NULL, estimated using k-nearest neighbors).
`minPts`	Integer, the minimum number of points required to form a dense region (default: NULL, computed as max(2 * ncol(x), 3)).
`scale`	Logical, whether to scale the data to have mean 0 and standard deviation 1 before clustering (default: TRUE).
`...`	Additional arguments passed to `dbscan::dbscan`.

Details

DBSCAN identifies clusters based on density, labeling points that do not belong to any dense cluster as noise (cluster 0), which are treated as rare events. If `eps` is not specified, it is estimated using the Kneedle algorithm of Satopaa et al. (2011) on the sorted k-nearest neighbor distances (where k = `minPts` - 1). If `minPts` is not specified, it is computed as max(2 * ncol(x), 3). Scaling is recommended to ensure features contribute equally to distance calculations. The `scores` are approximate, based on inverse k-nearest neighbor distances, and should be interpreted cautiously.

Value

A list containing:

- `scores`: Numeric vector of approximate anomaly scores (inverse of k-nearest neighbor distances, normalized to [0, 1]; higher values indicate more likely anomalies). Note: These are heuristic scores and may not align perfectly with DBSCAN's clustering decisions.
- `is_anomaly`: Logical vector indicating whether each observation is a rare event (TRUE for noise points, cluster label 0).
- `cluster`: Integer vector of cluster assignments (0 for noise, 1+ for clusters).
- `model`: The fitted DBSCAN model object.

References

Ester M, Kriegel H, Sander J, Xu X (1996). "A density-based algorithm for discovering clusters in large spatial databases with noise." In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, 226–231. https://dl.acm.org/doi/10.5555/3001460.3001507.

Satopaa V, Albrecht J, Irwin D, Raghavan B (2011). "Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior." In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, 166–171. doi:10.1109/ICDCSW.2011.20.

See Also

rare_iforest, rare_residuals, kneedle

Examples

```
set.seed(123)
data <- matrix(rnorm(1000), nrow = 500)
data[1:5, ] <- data[1:5, ] + 10  # Add some outliers
result <- rare_dbscan(data)
table(result$is_anomaly)
plot(data, col = ifelse(result$is_anomaly, "red", "blue"), pch = 19)

result <- rare_dbscan(iris[, 1:4], minPts = 20)
table(result$is_anomaly)

result <- rare_dbscan(swiss)
table(result$is_anomaly)
```

---

rare_heatwaves *Detect Heatwaves (or Cold Spells) Using heatwaveR*

Description

This is a thin wrapper around heatwaveR that detects discrete, prolonged, anomalous high-temperature events (Hobday et al. 2016). It constructs a seasonal climatology and a percentile-based threshold, and then identifies events that exceed the threshold for a minimum duration.

Usage

```
rare_heatwaves(
  data,
  date_col = "date",
  value_col = "value",
  climatology_period = NULL,
  pctile = 90,
  min_duration = 5,
  cold_spells = FALSE,
  ...
)
```

Arguments

`data`	A data.frame (or tibble) containing a Date column and a numeric temperature column.
`date_col`	Name of the Date column (default: "date").
`value_col`	Name of the numeric temperature column (default: "value").
`climatology_period`	Length-2 character vector with start and end dates (inclusive) for the fixed long-term climatology period, e.g., c("1981-01-01","2010-12-31"). Defaults to Hobday-style 30-year period if present in the data; otherwise the range of available dates will be used.
`pctile`	Percentile threshold for events (default: 90 for 90th percentile).
`min_duration`	Minimum consecutive days above threshold to define an event (default: 5).
`cold_spells`	Logical; if TRUE, detect cold-spells (events below the threshold) instead of heatwaves (default: FALSE).
`...`	Additional arguments passed to heatwaveR::ts2clm() or heatwaveR::detect_event().

Details

This function uses heatwaveR::ts2clm() to compute a seasonal climatology and a percentile-based threshold over a fixed climatology window, then heatwaveR::detect_event() to detect events. It assumes daily data. If your data are not daily, aggregate or resample to daily before calling this function to ensure meaningful climatologies and durations in days.

Value

A list with class "rare_heatwaves" containing:

data A data.frame with the full time series, seasonal climatology, threshold, and event indicators (follows heatwaveR column naming conventions).

events A data.frame summarising detected events (start, peak, end, duration, intensities, rates, etc.).

params A list of key parameters used: pctile, min_duration, climatology_period, and cold_spells.

References

Hobday AJ, Alexander LV, Perkins SE, Smale DA, Straub SC, Oliver ECJ, Benthuysen JA, Burrows MT, Donat MG, Feng M, N. J., Moore PJ, Scannell HA, Gupta AS, Wernberg T (2016). "A hierarchical approach to defining marine heatwaves." *Progress in Oceanography*, **141**, 227–238. doi:10.1016/j.pocean.2015.12.014.

Examples

```
# Example with Annapolis data
data(Annapolis)
summary(Annapolis)

# Detect heatwaves
hw <- rare_heatwaves(Annapolis, date_col = "date", value_col = "tmax",
```

```
                    cold_spells = FALSE,
                    pctile = 90, min_duration = 5)

# Detected events
head(hw$events)
tail(hw$events)

# Flagged days in the full series
table(hw$data$event)

# Detect cold spells
cs <- rare_heatwaves(Annapolis, date_col = "date", value_col = "tmax",
                    cold_spells = TRUE,
                    pctile = 10, min_duration = 5)

# Detected events
head(cs$events)
tail(cs$events)

# Flagged days in the full series
table(cs$data$event)
```

`rare_iforest` *Detect Rare Events Using Isolation Forest*

Description

This function applies an isolation forest algorithm (Liu et al. 2008) in a dataset using the `isotree` package. It fits a model, computes anomaly scores, and classifies observations based on a specified threshold.

Usage

```
rare_iforest(
  x,
  ndim = 1,
  ntrees = 500,
  missing_action = "fail",
  scoring_metric = "adj_depth",
  penalize_range = TRUE,
  nthreads = parallel::detectCores() - 1,
  threshold = 0.5,
  sample_size = NULL,
  ...
)
```

Arguments

x	A numeric matrix or data frame with no missing values (unless missing_action is set to handle them). Rows are observations, and columns are features.
ndim	Integer, number of dimensions to split at each node (default: 1). Higher values may improve detection for multivariate anomalies.
ntrees	Integer, number of trees in the forest (default: 500). More trees generally improve stability but increase computation time.
missing_action	Character, how to handle missing values: 'fail', 'impute', or 'auto' (default: 'fail').
scoring_metric	Character, scoring method for anomalies: 'depth', 'adj_depth', or 'density' (default: 'adj_depth').
penalize_range	Logical, whether to penalize features with smaller ranges (default: TRUE).
nthreads	Integer, number of threads for parallel processing (default: parallel::detectCores() - 1).
threshold	Numeric, anomaly score threshold for classifying rare events (default: 0.5). Scores above this value are classified as anomalies. Higher values are more conservative.
sample_size	Integer or fraction, number or proportion of rows to sample for training (default: NULL, uses all rows). If <= 1, interpreted as a fraction; otherwise as an absolute count.
...	Additional arguments passed to isotree::isolation.forest.

Value

A list containing:

- scores: Numeric vector of anomaly scores for each observation.
- is_anomaly: Logical vector indicating whether each observation is an anomaly (score > threshold).
- model: The fitted isotree isolation forest model.

References

Liu FT, Ting KM, Zhou Z (2008). "Isolation Forest." In *2008 Eighth IEEE International Conference on Data Mining*, 413–422. doi:10.1109/ICDM.2008.17.

See Also

rare_dbscan, rare_residuals

Examples

```
set.seed(123)
data <- matrix(rnorm(1000), nrow = 500)
data[1:5, ] <- data[1:5, ] + 10  # Add some outliers
result <- rare_iforest(data, ntrees = 100, threshold = 0.7)
```

```
table(result$is_anomaly)
plot(data, col = ifelse(result$is_anomaly, "red", "blue"), pch = 19)

result <- rare_iforest(iris[, 1:4], threshold = 0.5)
table(result$is_anomaly)

result <- rare_iforest(swiss, threshold = 0.5)
table(result$is_anomaly)
```

`rare_residuals` *Detect Rare Events in Time Series Residuals*

Description

This function filters a time series to compute residuals using STL decomposition (for seasonal data) or LOESS smoothing (for non-seasonal data) and identifies rare events (anomalies) in the residuals using Isolation Forest or DBSCAN. For seasonal time series, it applies STL decomposition and full-length Fourier smoothing separately to model periodic structure and estimate residuals.

Usage

```
rare_residuals(
  x,
  seasonal = FALSE,
  period = NULL,
  method = c("all", "iforest", "dbscan"),
  fourier_terms = 2,
  stl_args = list(),
  loess_args = list(),
  climatology_period = NULL,
  iforest_args = list(),
  dbscan_args = list()
)
```

Arguments

`x`	A numeric vector (time series values), a univariate `ts` object, or a data frame with columns `time` (numeric or Date) and `value` (numeric). If a vector, assumes regular time steps starting from 1. If a `ts` is provided, the time index is derived via `stats::time(x)`.
`seasonal`	Logical, indicating if the time series is seasonal (TRUE) or non-seasonal (FALSE) (default: FALSE). If `x` is a univariate `ts` object with `frequency(x) > 1`, the function will automatically treat the series as seasonal (i.e., set `seasonal = TRUE`).

`period` Numeric, the period of seasonality, e.g., 12 for monthly or 365.25 for daily data (default: NULL). If NULL and `seasonal == TRUE`, the period is automatically determined. For `ts` input, `frequency(x)` is used; otherwise, it is estimated using spectral analysis.

`method` Character, specifying the anomaly detection method: "iforest", "dbscan", or "all" (default: "all").

`fourier_terms` Integer, number of Fourier term pairs for seasonal modeling in Fourier smoothing (default: 2).

`stl_args` A named list of arguments passed to `stats::stl()` for detrending and deseasonalizing `x`. Default: empty list `list()`. If not provided or empty, dynamic defaults for stable seasonal decomposition are used: `s.window` is the nearest odd to `max(7, 1.5 * period)`, `t.window` is the nearest odd to `max(13, 1.5 * period)`, and `robust = TRUE`. If you provide a partial list, missing keys are filled with these dynamic defaults.

`loess_args` A named list of arguments passed to `stats::loess()` for non-seasonal smoothing. Default: `list()`. If not provided or empty, defaults `span = 0.75` and `degree = 1` are used. If you provide a partial list, missing keys are filled with these defaults. The formula is set to `value ~ time` unless you explicitly provide `formula`.

`climatology_period` Length-2 character vector with start and end dates (inclusive) for the climatology period used to estimate Fourier decomposition in seasonal data, e.g., c("1981-01-01","2010-12-31"). Only used when `seasonal = TRUE` and `time` is Date. Defaults to a 30-year period if available in the data; otherwise the full range of available dates is used.

`iforest_args` A named list of arguments passed to `rare_iforest()` (e.g., `list(ntrees = 200)`). Default: `list()`.

`dbscan_args` A named list of arguments passed to `rare_dbscan()` (e.g., `list(minPts = 10)`). Default: `list()`.

Details

For non-seasonal data, residuals are computed using LOESS with `time` as the predictor. For seasonal data, residuals are computed twice: (1) using STL decomposition to extract the remainder component, and (2) using a full-length Fourier series to capture fixed periodicity. The STL seasonal component is smoothed using the `s.window` parameter (numeric for flexible smoothing, "periodic" for fixed seasonality). Rare events are detected in STL (or LOESS for non-seasonal) residuals using `rare_iforest` or `rare_dbscan`. The period is estimated via spectral analysis if not provided. Input validation prevents coercion errors. Separate argument lists (`stl_args`, `iforest_args`, `dbscan_args`) ensure function-specific parameters are passed correctly.

For seasonal data with Date-based time indices, the `climatology_period` parameter allows specification of a reference period for estimating the Fourier seasonal cycle. The Fourier model is fitted using only data within this period. This is useful for detecting changes in seasonality patterns relative to a baseline climatology.

If `x` is a univariate `ts` object with `frequency(x) > 1`, the function will automatically treat the series as seasonal (i.e., set `seasonal = TRUE`). When `period` is not provided, `frequency(x)` will be used.

Value

A list containing:

- `data`: A data frame with:
  - `time`: Input time points.
  - `value`: Original time series values.
  - `residual`: Residuals from STL (seasonal) or LOESS (non-seasonal) smoothing.
  - `residual_fourier`: Residuals from Fourier smoothing (if `seasonal = TRUE`).
  - `is_anomaly_iforest`: Logical, anomalies from Isolation Forest (if applicable).
  - `is_anomaly_dbscan`: Logical, anomalies from DBSCAN (if applicable).
  - `score_iforest`: Anomaly scores from Isolation Forest.
  - `score_dbscan`: Anomaly scores from DBSCAN.
- `params`: A list of parameters used, including `period`, `climatology_period` (if seasonal data with Date time index), and `method`.

See Also

`rare_iforest`, `rare_dbscan`, `mcc`, `stl`

Examples

```
# Annapolis temperature (seasonal, Date index)
data(Annapolis)
x <- data.frame(time = Annapolis$date, value = Annapolis$tmax)
result_tmax <- rare_residuals(x,
                          seasonal = TRUE, period = 365.25,
                          method = "iforest",
                          iforest_args = list(ntrees = 100))
plot(result_tmax$data$time, result_tmax$data$value, type = "l", las = 1,
     xlab = "Date", ylab = "Temperature (°C)",
     main = "Anomalies in Annapolis daily maximum temperature",
)
anomaly_sign <- sign(result_tmax$data$residual[result_tmax$data$is_anomaly_iforest])
points(result_tmax$data$time[result_tmax$data$is_anomaly_iforest],
       result_tmax$data$value[result_tmax$data$is_anomaly_iforest],
       col = c("blue", NA, "red")[anomaly_sign + 2],
       pch = 19)

# Annapolis annual fish catch example (hypothetical data)
years <- 1990:2024
# Annual fish catch (tonnes)
fish_catch <- c(505, 493, 498, 487, 450, 479, 488, 472, 465, 476,
                458, 466, 453, 461, 500, 447, 441, 435, 428, 443,
                425, 418, 432, 421, 345, 412, 407, 418, 405, 397,
                411, 403, 395, 408, 391)
result_fish <- rare_residuals(data.frame(time = years, value = fish_catch),
                              method = "iforest",
                              iforest_args = list(ntrees = 100))
```

```
# Aggregate Annapolis daily data to identify rare warm spring years
# and test association with rare fish-catch years
spring_months <- c(3, 4, 5)
rare_tmax_spring <- result_tmax$data |>
    dplyr::mutate(Year = format(time, "%Y"),
                  Month = as.numeric(format(time, "%m"))) |>
    dplyr::filter(Month %in% spring_months, Year %in% years) |>
    dplyr::group_by(Year) |>
    dplyr::summarise(rare_tmax_spring = any(is_anomaly_iforest & residual > 0)) |>
    dplyr::arrange(Year) |>
    dplyr::pull(rare_tmax_spring)

# Test association between rare fish-catch years and rare warm years
mcc_result <- mcc(result_fish$data$is_anomaly_iforest,
                  rare_tmax_spring,
                  ts = TRUE, bootstrap_reps = 999)
mcc_result$mcc
mcc_result$mcc_bootstrap_pv

# Classic data example (ts input)
result_AirPassengers <- rare_residuals(AirPassengers,
                                       method = "iforest",
                                       iforest_args = list(ntrees = 100,
                                                           threshold = 0.6))
# View results
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(ggplot2)
  ggplot2::ggplot(result_AirPassengers$data, aes(x = time, y = value)) +
     geom_line(color = "gray") +
     geom_point(aes(color = is_anomaly_iforest)) +
     labs(title = "Anomaly detection in AirPassengers using isolation forest",
          x = "Time",
          y = "Number of passengers",
          color = "Anomaly status") +
     scale_color_manual(values = c("gray", "red"), labels = c("Normal", "Anomaly")) +
     theme_minimal()
}
```

Index