Package ‘timetk’

July 22, 2025

Type Package

Title A Tool Kit for Working with Time Series

Version 2.9.0

Description Easy visualization, wrangling, and feature engineering of time series data for forecasting and machine learning prediction. Consolidates and extends time series functionality from packages including 'dplyr', 'stats', 'xts', 'forecast', 'slider', 'padr', 'recipes', and 'rsample'.

URL https://github.com/business-science/timetk, https://business-science.github.io/timetk/

BugReports https://github.com/business-science/timetk/issues

License GPL (>= 3)

Encoding UTF-8

LazyData true

Depends R (>= 3.3.0)

Imports recipes (>= 1.0.4), rsample, dplyr (>= 1.0.0), ggplot2 (>= 3.4.0), forcats, stringr, plotly, lubridate (>= 1.6.0), padr (>= 0.5.2), purrr (>= 0.2.2), readr (>= 1.3.0), stringi (>= 1.4.6), tibble (>= 3.0.3), tidyr (>= 1.1.0), xts (>= 0.9-7), zoo (>= 1.7-14), rlang (>= 1.1.1), tidyselect (>= 1.1.0), slider, anytime, timeDate, forecast, tsfeatures, hms, generics

Suggests modeltime, glmnet, workflows, parsnip, tune (>= 0.1.2), knitr, rmarkdown, broom, scales, testthat, fracdiff, timeSeries, tseries, trelliscopejs

RoxygenNote 7.2.3

VignetteBuilder knitr

NeedsCompilation no

Author Matt Dancho [aut, cre], Davis Vaughan [aut]

Maintainer Matt Dancho <mdancho@business-science.io>

Repository CRAN

Date/Publication 2023-10-31 22:30:02 UTC

Contents

timetk-package *timetk: Time Series Analysis in the Tidyverse*

Description

The `timetk` package combines a collection of coercion tools for time series analysis.

Details

The `timetk` package has several benefits:

1. Visualizing Time Series
2. Wrangling Time Series.
3. Preprocessing and Feature Engineering.

To learn more about `timetk`, start with the documentation: https://business-science.github.io/timetk/

Author(s)

Maintainer: Matt Dancho `<mdancho@business-science.io>`

Authors:

- Davis Vaughan `<dvaughan@business-science.io>`

See Also

Useful links:

- https://github.com/business-science/timetk
- https://business-science.github.io/timetk/
- Report bugs at https://github.com/business-science/timetk/issues

anomalize *Automatic group-wise Anomaly Detection*

Description

`anomalize()` is used to detect anomalies in time series data, either for a single time series or for multiple time series grouped by a specific column.

Usage

```
anomalize(
  .data,
  .date_var,
  .value,
  .frequency = "auto",
  .trend = "auto",
  .method = "stl",
  .iqr_alpha = 0.05,
  .clean_alpha = 0.75,
  .max_anomalies = 0.2,
  .message = TRUE
)
```

Arguments

`.data`	A `tibble` or `data.frame` with a time-based column
`.date_var`	A column containing either date or date-time values
`.value`	A column containing numeric values
`.frequency`	Controls the seasonal adjustment (removal of seasonality). Input can be either "auto", a time-based definition (e.g. "2 weeks"), or a numeric number of observations per frequency (e.g. 10). Refer to `tk_get_frequency()`.
`.trend`	Controls the trend component. For STL, trend controls the sensitivity of the LOESS smoother, which is used to remove the remainder. Refer to `tk_get_trend()`.
`.method`	The outlier detection method. Default: "stl". Currently "stl" is the only method. "twitter" is planned.
`.iqr_alpha`	Controls the width of the "normal" range. Lower values are more conservative while higher values are less prone to incorrectly classifying "normal" observations.
`.clean_alpha`	Controls the threshold for cleaning the outliers. The default is 0.75, which means that the anomalies will be cleaned using the 0.75 * lower or upper bound of the recomposed time series, depending on the direction of the anomaly.
`.max_anomalies`	The maximum percent of anomalies permitted to be identified.
`.message`	A boolean. If `TRUE`, will output information related to automatic frequency and trend selection (if applicable).

Details

The `anomalize()` method for anomaly detection that implements a 2-step process to detect outliers in time series.

Step 1: Detrend & Remove Seasonality using STL Decomposition

The decomposition separates the "season" and "trend" components from the "observed" values leaving the "remainder" for anomaly detection.

The user can control two parameters: frequency and trend.

1. `.frequency`: Adjusts the "season" component that is removed from the "observed" values.
2. `.trend`: Adjusts the trend window (t.window parameter from `stats::stl()` that is used.

The user may supply both `.frequency` and `.trend` as time-based durations (e.g. "6 weeks") or numeric values (e.g. 180) or "auto", which predetermines the frequency and/or trend based on the scale of the time series using the `tk_time_scale_template()`.

Step 2: Anomaly Detection

Once "trend" and "season" (seasonality) is removed, anomaly detection is performed on the "remainder". Anomalies are identified, and boundaries (recomposed_l1 and recomposed_l2) are determined.

The Anomaly Detection Method uses an inner quartile range (IQR) of +/-25 the median.

IQR Adjustment, alpha parameter

With the default `alpha = 0.05`, the limits are established by expanding the 25/75 baseline by an IQR Factor of 3 (3X). The *IQR Factor = 0.15 / alpha* (hence 3X with alpha = 0.05):

- To increase the IQR Factor controlling the limits, decrease the alpha, which makes it more difficult to be an outlier.
- Increase alpha to make it easier to be an outlier.
- The IQR outlier detection method is used in `forecast::tsoutliers()`.
- A similar outlier detection method is used by Twitter's `AnomalyDetection` package.
- Both Twitter and Forecast tsoutliers methods have been implemented in Business Science's `anomalize` package.

Value

A `tibble` or `data.frame` with the following columns:

- observed: original data
- seasonal: seasonal component
- seasadaj: seasonal adjusted
- trend: trend component
- remainder: residual component
- anomaly: Yes/No flag for outlier detection
- anomaly score: distance from centerline
- anomaly direction: -1, 0, 1 inidicator for direction of the anomaly
- recomposed_l1: lower level bound of recomposed time series
- recomposed_l2: upper level bound of recomposed time series
- observed_clean: original data with anomalies interpolated

References

1. CLEVELAND, R. B., CLEVELAND, W. S., MCRAE, J. E., AND TERPENNING, I. STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Journal of Official Statistics, Vol. 6, No. 1 (1990), pp. 3-73.
2. Owen S. Vallis, Jordan Hochenbaum and Arun Kejariwal (2014). A Novel Technique for Long-Term Anomaly Detection in the Cloud. Twitter Inc.

Examples

```
library(dplyr)

walmart_sales_weekly %>%
    filter(id %in% c("1_1", "1_3")) %>%
    group_by(id) %>%
    anomalize(Date, Weekly_Sales)
```

`between_time` *Between (For Time Series): Range detection for date or date-time sequences*

Description

The easiest way to filter time series date or date-time vectors. Returns a logical vector indicating which date or date-time values are within a range. See `filter_by_time()` for the `data.frame` (`tibble`) implementation.

Usage

```
between_time(index, start_date = "start", end_date = "end")
```

Arguments

`index`	A date or date-time vector.
`start_date`	The starting date
`end_date`	The ending date

Details

Pure Time Series Filtering Flexibilty

The `start_date` and `end_date` parameters are designed with flexibility in mind.

Each side of the `time_formula` is specified as the character `'YYYY-MM-DD HH:MM:SS'`, but powerful shorthand is available. Some examples are:

- **Year:** `start_date = '2013', end_date = '2015'`
- **Month:** `start_date = '2013-01', end_date = '2016-06'`
- **Day:** `start_date = '2013-01-05', end_date = '2016-06-04'`
- **Second:** `start_date = '2013-01-05 10:22:15', end_date = '2018-06-03 12:14:22'`
- **Variations:** `start_date = '2013', end_date = '2016-06'`

Key Words: "start" and "end"

Use the keywords "start" and "end" as shorthand, instead of specifying the actual start and end values. Here are some examples:

- **Start of the series to end of 2015:** `start_date = 'start', end_date = '2015'`
- **Start of 2014 to end of series:** `start_date = '2014', end_date = 'end'`

Internal Calculations

All shorthand dates are expanded:

- The `start_date` is expanded to be the *first date* in that period
- The `end_date` side is expanded to be the *last date* in that period

This means that the following examples are equivalent (assuming your index is a POSIXct):

- `start_date = '2015'` is equivalent to `start_date = '2015-01-01 + 00:00:00'`
- `end_date = '2016'` is equivalent to `2016-12-31 + 23:59:59'`

Value

A `logical` vector the same length as `index` indicating whether or not the timestamp value was within the `start_date` and `end_date` range.

References

- This function is based on the `tibbletime::filter_time()` function developed by Davis Vaughan.

See Also

Time-Based dplyr functions:

- `summarise_by_time()` - Easily summarise using a date column.
- `mutate_by_time()` - Simplifies applying mutations by time windows.
- `pad_by_time()` - Insert time series rows with regularly spaced timestamps
- `filter_by_time()` - Quickly filter using date ranges.
- `filter_period()` - Apply filtering expressions inside periods (windows)
- `slice_period()` - Apply slice inside periods (windows)
- `condense_period()` - Convert to a different periodicity
- `between_time()` - Range detection for date or date-time sequences.
- `slidify()` - Turn any function into a sliding (rolling) function

Examples

```
library(dplyr)

index_daily <- tk_make_timeseries("2016-01-01", "2017-01-01", by = "day")
index_min   <- tk_make_timeseries("2016-01-01", "2017-01-01", by = "min")

# How it works
# - Returns TRUE/FALSE length of index
# - Use sum() to tally the number of TRUE values
```

```
index_daily %>% between_time("start", "2016-01") %>% sum()

# ---- INDEX SLICING ----

# Daily Series: Month of January 2016
index_daily[index_daily %>% between_time("start", "2016-01")]

# Daily Series: March 1st - June 15th, 2016
index_daily[index_daily %>% between_time("2016-03", "2016-06-15")]

# Minute Series:
index_min[index_min %>% between_time("2016-02-01 12:00", "2016-02-01 13:00")]

# ---- FILTERING WITH DPLYR ----
FANG %>%
    group_by(symbol) %>%
    filter(date %>% between_time("2016-01", "2016-01"))
```

---

bike_sharing_daily *Daily Bike Sharing Data*

Description

This dataset contains the daily count of rental bike transactions between years 2011 and 2012 in Capital bikeshare system with the corresponding weather and seasonal information.

Usage

```
bike_sharing_daily
```

Format

A tibble: 731 x 16

- instant: record index
- dteday : date
- season : season (1:winter, 2:spring, 3:summer, 4:fall)
- yr : year (0: 2011, 1:2012)
- mnth : month (1 to 12)
- hr : hour (0 to 23)
- holiday : weather day is holiday or not
- weekday : day of the week
- workingday : if day is neither weekend nor holiday is 1, otherwise is 0.
- weathersit :
  - 1: Clear, Few clouds, Partly cloudy, Partly cloudy

  - 2: Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
  - 3: Light Snow, Light Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
  - 4: Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
- temp : Normalized temperature in Celsius. The values are derived via (t-t_min)/(t_max-t_min), t_min=-8, t_max=+39 (only in hourly scale)
- atemp: Normalized feeling temperature in Celsius. The values are derived via (t-t_min)/(t_max-t_min), t_min=-16, t_max=+50 (only in hourly scale)
- hum: Normalized humidity. The values are divided to 100 (max)
- windspeed: Normalized wind speed. The values are divided to 67 (max)
- casual: count of casual users
- registered: count of registered users
- cnt: count of total rental bikes including both casual and registered

References

Fanaee-T, Hadi, and Gama, Joao, 'Event labeling combining ensemble detectors and background knowledge', Progress in Artificial Intelligence (2013): pp. 1-15, Springer Berlin Heidelberg.

Examples

```
bike_sharing_daily
```

---

box_cox_vec *Box Cox Transformation*

Description

This is mainly a wrapper for the BoxCox transformation from the `forecast` R package. The `box_cox_vec()` function performs the transformation. `box_cox_inv_vec()` inverts the transformation. `auto_lambda()` helps in selecting the optimal `lambda` value.

Usage

```
box_cox_vec(x, lambda = "auto", silent = FALSE)

box_cox_inv_vec(x, lambda)

auto_lambda(
  x,
  method = c("guerrero", "loglik"),
  lambda_lower = -1,
  lambda_upper = 2
)
```

Arguments

`x`	A numeric vector.
`lambda`	The box cox transformation parameter. If set to "auto", performs automated lambda selection using `auto_lambda()`.
`silent`	Whether or not to report the automated `lambda` selection as a message.
`method`	The method used for automatic `lambda` selection. Either "guerrero" or "loglik".
`lambda_lower`	A lower limit for automatic `lambda` selection
`lambda_upper`	An upper limit for automatic `lambda` selection

Details

The Box Cox transformation is a power transformation that is commonly used to reduce variance of a time series.

Automatic Lambda Selection

If desired, the `lambda` argument can be selected using `auto_lambda()`, a wrapper for the Forecast R Package's `forecast::BoxCox.lambda()` function. Use either of 2 methods:

1. "guerrero" - Minimizes the non-seasonal variance
2. "loglik" - Maximizes the log-likelihood of a linear model fit to `x`

Value

Returns a `numeric` vector that has been transformed.

References

- Forecast R Package
- Forecasting: Principles & Practices: Transformations & Adjustments
- Guerrero, V.M. (1993) Time-series analysis supported by power transformations. *Journal of Forecasting*, 12, 37–48.

See Also

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`, `ts_clean_vec()`

Other common transformations to reduce variance: `log()`, `log1p()` and `sqrt()`

Examples

```
library(dplyr)
d10_daily <- m4_daily %>% dplyr::filter(id == "D10")

# --- VECTOR ----

value_bc <- box_cox_vec(d10_daily$value)
value    <- box_cox_inv_vec(value_bc, lambda = 1.25119350454964)

# --- MUTATE ----

m4_daily %>%
    dplyr::group_by(id) %>%
    dplyr::mutate(value_bc = box_cox_vec(value))
```

---

condense_period *Convert the Period to a Lower Periodicity (e.g. Go from Daily to Monthly)*

Description

Convert a `data.frame` object from daily to monthly, from minute data to hourly, and more. This allows the user to easily aggregate data to a less granular level by taking the value from either the beginning or end of the period.

Usage

```
condense_period(.data, .date_var, .period = "1 day", .side = c("start", "end"))
```

Arguments

`.data` A `tbl` object or `data.frame`

`.date_var` A column containing date or date-time values. If missing, attempts to auto-detect date column.

`.period` A period to condense the time series to. Time units are condensed using `lubridate::floor_date()` or `lubridate::ceiling_date()`.

The value can be:

- `second`
- `minute`
- `hour`
- `day`
- `week`
- `month`
- `bimonth`
- `quarter`

- season
- halfyear
- year

Arbitrary unique English abbreviations as in the `lubridate::period()` constructor are allowed:

- "1 year"
- "2 months"
- "30 seconds"

`.side` One of "start" or "end". Determines if the first observation in the period should be returned or the last.

Value

A `tibble` or `data.frame`

See Also

Time-Based dplyr functions:

- `summarise_by_time()` - Easily summarise using a date column.
- `mutate_by_time()` - Simplifies applying mutations by time windows.
- `pad_by_time()` - Insert time series rows with regularly spaced timestamps
- `filter_by_time()` - Quickly filter using date ranges.
- `filter_period()` - Apply filtering expressions inside periods (windows)
- `slice_period()` - Apply slice inside periods (windows)
- `condense_period()` - Convert to a different periodicity
- `between_time()` - Range detection for date or date-time sequences.
- `slidify()` - Turn any function into a sliding (rolling) function

Examples

```
# Libraries
library(dplyr)

# First value in each month
m4_daily %>%
    group_by(id) %>%
    condense_period(.period = "1 month")

# Last value in each month
m4_daily %>%
    group_by(id) %>%
    condense_period(.period = "1 month", .side = "end")
```

`diff_vec` *Differencing Transformation*

Description

`diff_vec()` applies a Differencing Transformation. `diff_inv_vec()` inverts the differencing transformation.

Usage

```
diff_vec(
  x,
  lag = 1,
  difference = 1,
  log = FALSE,
  initial_values = NULL,
  silent = FALSE
)

diff_inv_vec(x, lag = 1, difference = 1, log = FALSE, initial_values = NULL)
```

Arguments

`x`	A numeric vector to be differenced or inverted.
`lag`	Which lag (how far back) to be included in the differencing calculation.
`difference`	The number of differences to perform. • 1 Difference is equivalent to measuring period change. • 2 Differences is equivalent to measuring period acceleration.
`log`	If log differences should be calculated. *Note that difference inversion of a log-difference is approximate.*
`initial_values`	Only used in the `diff_vec_inv()` operation. A numeric vector of the initial values, which are used to invert differences. This vector is the original values that are the length of the `NA` missing differences.
`silent`	Whether or not to report the initial values used to invert the difference as a message.

Details

Benefits:

This function is `NA` padded by default so it works well with `dplyr::mutate()` operations.

Difference Calculation

Single differencing, `diff_vec(x_t)` is equivalent to: `x_t - x_t1`, where the subscript _t1 indicates the first lag. *This transformation can be interpereted as change.*

Double Differencing Calculation

Double differencing, `diff_vec(x_t, difference = 2)` is equivalent to: `(x_t - x_t1) - (x_t - x_t1)_t1`, where the subscript _t1 indicates the first lag. *This transformation can be interpereted as acceleration.*

Log Difference Calculation

Log differencing, `diff_vec(x_t, log = TRUE)` is equivalent to: `log(x_t) - log(x_t1) = log(x_t / x_t1)`, where x_t is the series and x_t1 is the first lag.

The 1st difference `diff_vec(difference = 1, log = TRUE)` has an interesting property where `diff_vec(difference = 1, log = TRUE) %>% exp()` is approximately *1 + rate of change.*

Value

A numeric vector

See Also

Advanced Differencing and Modeling:

- `step_diff()` - Recipe for `tidymodels` workflow
- `tk_augment_differences()` - Adds many differences to a `data.frame` (`tibble`)

Additional Vector Functions:

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`, `ts_clean_vec()`

Examples

```
library(dplyr)

# --- USAGE ----

diff_vec(1:10, lag = 2, difference = 2) %>%
    diff_inv_vec(lag = 2, difference = 2, initial_values = 1:4)

# --- VECTOR ----

# Get Change
1:10 %>% diff_vec()

# Get Acceleration
1:10 %>% diff_vec(difference = 2)

# Get approximate rate of change
```

```
1:10 %>% diff_vec(log = TRUE) %>% exp() - 1


# --- MUTATE ----

m4_daily %>%
    group_by(id) %>%
    mutate(difference = diff_vec(value, lag = 1)) %>%
    mutate(
        difference_inv = diff_inv_vec(
            difference,
            lag = 1,
            # Add initial value to calculate the inverse difference
            initial_values = value[1]
        )
    )
```

FANG *Stock prices for the "FANG" stocks.*

Description

A dataset containing the daily historical stock prices for the "FANG" tech stocks, "FB", "AMZN", "NFLX", and "GOOG", spanning from the beginning of 2013 through the end of 2016.

Usage

```
FANG
```

Format

A "tibble" ("tidy" data frame) with 4,032 rows and 8 variables:

symbol stock ticker symbol

date trade date

open stock price at the open of trading, in USD

high stock price at the highest point during trading, in USD

low stock price at the lowest point during trading, in USD

close stock price at the close of trading, in USD

volume number of shares traded

adjusted stock price at the close of trading adjusted for stock splits, in USD `filter_by_time` *Filter (for Time-Series Data)*

Description

The easiest way to filter time-based **start/end ranges** using shorthand timeseries notation. See `filter_period()` for applying filter expression by period (windows).

Usage

```
filter_by_time(.data, .date_var, .start_date = "start", .end_date = "end")
```

Arguments

`.data`	A tibble with a time-based column.
`.date_var`	A column containing date or date-time values to filter. If missing, attempts to auto-detect date column.
`.start_date`	The starting date for the filter sequence
`.end_date`	The ending date for the filter sequence

Details

Pure Time Series Filtering Flexibilty

The `.start_date` and `.end_date` parameters are designed with flexibility in mind.

Each side of the `time_formula` is specified as the character `'YYYY-MM-DD HH:MM:SS'`, but powerful shorthand is available. Some examples are:

- **Year:** `.start_date = '2013', .end_date = '2015'`
- **Month:** `.start_date = '2013-01', .end_date = '2016-06'`
- **Day:** `.start_date = '2013-01-05', .end_date = '2016-06-04'`
- **Second:** `.start_date = '2013-01-05 10:22:15', .end_date = '2018-06-03 12:14:22'`
- **Variations:** `.start_date = '2013', .end_date = '2016-06'`

Key Words: "start" and "end"

Use the keywords "start" and "end" as shorthand, instead of specifying the actual start and end values. Here are some examples:

- **Start of the series to end of 2015:** `.start_date = 'start', .end_date = '2015'`
- **Start of 2014 to end of series:** `.start_date = '2014', .end_date = 'end'`

Internal Calculations

All shorthand dates are expanded:

- The `.start_date` is expanded to be the *first date* in that period
- The `.end_date` side is expanded to be the *last date* in that period

This means that the following examples are equivalent (assuming your index is a POSIXct):

- `.start_date = '2015'` is equivalent to `.start_date = '2015-01-01 + 00:00:00'`
- `.end_date = '2016'` is equivalent to `2016-12-31 + 23:59:59'`

Value

Returns a `tibble` or `data.frame` that has been filtered.

References

- This function is based on the `tibbletime::filter_time()` function developed by Davis Vaughan.

See Also

Time-Based dplyr functions:

- `summarise_by_time()` - Easily summarise using a date column.
- `mutate_by_time()` - Simplifies applying mutations by time windows.
- `pad_by_time()` - Insert time series rows with regularly spaced timestamps
- `filter_by_time()` - Quickly filter using date ranges.
- `filter_period()` - Apply filtering expressions inside periods (windows)
- `slice_period()` - Apply slice inside periods (windows)
- `condense_period()` - Convert to a different periodicity
- `between_time()` - Range detection for date or date-time sequences.
- `slidify()` - Turn any function into a sliding (rolling) function

Examples

```
library(dplyr)

# Filter values in January 1st through end of February, 2013
FANG %>%
    group_by(symbol) %>%
    filter_by_time(.start_date = "start", .end_date = "2013-02") %>%
    plot_time_series(date, adjusted, .facet_ncol = 2, .interactive = FALSE)
```

filter_period *Apply filtering expressions inside periods (windows)*

Description

Applies a dplyr **filtering expression inside a time-based period (window).** See `filter_by_time()` for filtering continuous ranges defined by start/end dates. `filter_period()` enables filtering expressions like:

- Filtering to the maximum value each month.
- Filtering the first date each month.
- Filtering all rows with value greater than a monthly average

Usage

```
filter_period(.data, ..., .date_var, .period = "1 day")
```

Arguments

`.data` A `tbl` object or `data.frame`

`...` Filtering expression. Expressions that return a logical value, and are defined in terms of the variables in .data. If multiple expressions are included, they are combined with the & operator. Only rows for which all conditions evaluate to TRUE are kept.

`.date_var` A column containing date or date-time values. If missing, attempts to auto-detect date column.

`.period` A period to filter within. Time units are grouped using `lubridate::floor_date()` or `lubridate::ceiling_date()`.

The value can be:

- `second`
- `minute`
- `hour`
- `day`
- `week`
- `month`
- `bimonth`
- `quarter`
- `season`
- `halfyear`
- `year`

Arbitrary unique English abbreviations as in the `lubridate::period()` constructor are allowed:

- `"1 year"`
- `"2 months"`
- `"30 seconds"`

Value

A tibble or data.frame

See Also

Time-Based dplyr functions:

- summarise_by_time() - Easily summarise using a date column.
- mutate_by_time() - Simplifies applying mutations by time windows.
- pad_by_time() - Insert time series rows with regularly spaced timestamps
- filter_by_time() - Quickly filter using date ranges.
- filter_period() - Apply filtering expressions inside periods (windows)
- slice_period() - Apply slice inside periods (windows)
- condense_period() - Convert to a different periodicity
- between_time() - Range detection for date or date-time sequences.
- slidify() - Turn any function into a sliding (rolling) function

Examples

```
# Libraries
library(dplyr)

# Max value in each month
m4_daily %>%
    group_by(id) %>%
    filter_period(.period = "1 month", value == max(value))

# First date each month
m4_daily %>%
    group_by(id) %>%
    filter_period(.period = "1 month", date == first(date))

# All observations that are greater than a monthly average
m4_daily %>%
    group_by(id) %>%
    filter_period(.period = "1 month", value > mean(value))
```

---

fourier_vec *Fourier Series*

Description

fourier_vec() calculates a Fourier Series from a date or date-time index.

Usage

```
fourier_vec(x, period, K = 1, type = c("sin", "cos"), scale_factor = NULL)
```

Arguments

`x`	A date, POSIXct, yearmon, yearqtr, or numeric sequence (scaled to difference 1 for `period` alignment) to be converted to a fourier series.
`period`	The number of observations that complete one cycle.
`K`	The fourier term order.
`type`	Either "sin" or "cos" for the appropriate type of fourier term.
`scale_factor`	Scale factor is a calculated value that scales date sequences to numeric sequences. A user can provide a different value of scale factor to override the date scaling. Default: NULL (auto-scale).

Details

Benefits:

This function is `NA` padded by default so it works well with `dplyr::mutate()` operations.

Fourier Series Calculation

The internal calculation is relatively straightforward: `fourier(x) = sin(2 * pi * term * x) or cos(2 * pi * term * x)`, where `term = K / period`.

Period Alignment, period

The `period` alignment with the sequence is an essential part of fourier series calculation.

- **Date, Date-Time, and Zoo (yearqtr and yearmon) Sequences** - Are scaled to unit difference of 1. This happens internally, so there's nothing you need to do or to worry about. Future time series will be scaled appropriately.
- **Numeric Sequences** - Are not scaled, which means you should transform them to a unit difference of 1 so that your x is a sequence that increases by 1. Otherwise your period and fourier order will be incorrectly calculated. The solution is to just take your sequence and divide by the median difference between values.

Fourier Order, K

The fourier order is a parameter that increases the frequency. `K = 2` doubles the frequency. It's common in time series analysis to add multiple fourier orders (e.g. 1 through 5) to account for seasonalities that occur faster than the primary seasonality.

Type (Sin/Cos)

The type of the fourier series can be either `sin` or `cos`. It's common in time series analysis to add both sin and cos series.

Value

A numeric vector

See Also

Fourier Modeling Functions:

- step_fourier() - Recipe for tidymodels workflow
- tk_augment_fourier() - Adds many fourier series to a data.frame (tibble)

Additional Vector Functions:

- Fourier Series: fourier_vec()
- Box Cox Transformation: box_cox_vec()
- Lag Transformation: lag_vec()
- Differencing Transformation: diff_vec()
- Rolling Window Transformation: slidify_vec()
- Loess Smoothing Transformation: smooth_vec()
- Missing Value Imputation for Time Series: ts_impute_vec(), ts_clean_vec()

Examples

```
library(dplyr)

# Set max.print to 50
options_old <- options()$max.print
options(max.print = 50)

date_sequence <- tk_make_timeseries("2016-01-01", "2016-01-31", by = "hour")

# --- VECTOR ---

fourier_vec(date_sequence, period = 7 * 24, K = 1, type = "sin")

# --- MUTATE ---

tibble(date = date_sequence) %>%
    # Add cosine series that oscilates at a 7-day period
    mutate(
        C1_7 = fourier_vec(date, period = 7*24, K = 1, type = "cos"),
        C2_7 = fourier_vec(date, period = 7*24, K = 2, type = "cos")
    ) %>%
    # Visualize
    tidyr::pivot_longer(cols = contains("_"), names_to = "name", values_to = "value") %>%
    plot_time_series(
        date, value, .color_var = name,
        .smooth = FALSE,
        .interactive = FALSE,
        .title = "7-Day Fourier Terms"
    )

options(max.print = options_old)
```

future_frame — *Make future time series from existing*

Description

Make future time series from existing

Usage

```
future_frame(
  .data,
  .date_var,
  .length_out,
  .inspect_weekdays = FALSE,
  .inspect_months = FALSE,
  .skip_values = NULL,
  .insert_values = NULL,
  .bind_data = FALSE
)
```

Arguments

`.data`	A data.frame or tibble
`.date_var`	A date or date-time variable.
`.length_out`	Number of future observations. Can be numeric number or a phrase like "1 year".
`.inspect_weekdays`	Uses a logistic regression algorithm to inspect whether certain weekdays (e.g. weekends) should be excluded from the future dates. Default is `FALSE`.
`.inspect_months`	Uses a logistic regression algorithm to inspect whether certain days of months (e.g. last two weeks of year or seasonal days) should be excluded from the future dates. Default is `FALSE`.
`.skip_values`	A vector of same class as `idx` of timeseries values to skip.
`.insert_values`	A vector of same class as `idx` of timeseries values to insert.
`.bind_data`	Whether or not to perform a row-wise bind of the `.data` and the future data. Default: `FALSE`

Details

This is a wrapper for `tk_make_future_timeseries()` that works on data.frames. It respects `dplyr` groups.

Specifying Length of Future Observations

The argument `.length_out` determines how many future index observations to compute. It can be specified as:

- **A numeric value** - the number of future observations to return.
  - The number of observations returned is *always* equal to the value the user inputs.
  - The **end date can vary** based on the number of timestamps chosen.
- **A time-based phrase** - The duration into the future to include (e.g. "6 months" or "30 minutes").
  - The *duration* defines the *end date* for observations.
  - The **end date will not change** and those timestamps that fall within the end date will be returned (e.g. a quarterly time series will return 4 quarters if `.length_out = "1 year"`).
  - The number of observations will vary to fit within the end date.

Weekday and Month Inspection

The `.inspect_weekdays` and `.inspect_months` arguments apply to "daily" (scale = "day") data (refer to `tk_get_timeseries_summary()` to get the index scale).

- The `.inspect_weekdays` argument is useful in determining missing days of the week that occur on a weekly frequency such as every week, every other week, and so on. It's recommended to have at least 60 days to use this option.
- The `.inspect_months` argument is useful in determining missing days of the month, quarter or year; however, the algorithm can inadvertently select incorrect dates if the pattern is erratic.

Skipping / Inserting Values

The `.skip_values` and `.insert_values` arguments can be used to remove and add values into the series of future times. The values must be the same format as the `idx` class.

- The `.skip_values` argument useful for passing holidays or special index values that should be excluded from the future time series.
- The `.insert_values` argument is useful for adding values back that the algorithm may have excluded.

Binding with Data

Rowwise binding with the original is so common that I've added an argument `.bind_data` to perform a row-wise bind of the future data and the incoming data.

This *replaces* the need to do:

```
df %>%
   future_frame(.length_out = "6 months") %>%
   bind_rows(df, .)
```

Now you can just do:

```
df %>%
    future_frame(.length_out = "6 months", .bind_data = TRUE)
```

Value

A tibble that has been extended with future date, date-time timestamps.

See Also

- Making Future Time Series: tk_make_future_timeseries() (Underlying function)

Examples

```
library(dplyr)

# 30-min interval data
taylor_30_min %>%
    future_frame(date, .length_out = "1 week")

# Daily Data (Grouped)
m4_daily %>%
    group_by(id) %>%
    future_frame(date, .length_out = "6 weeks")

# Specify how many observations to project into the future
m4_daily %>%
    group_by(id) %>%
    future_frame(date, .length_out = 100)

# Bind with Original Data
m4_daily %>%
    group_by(id) %>%
    future_frame(date, .length_out = 100, .bind_data = TRUE)

holidays <- tk_make_holiday_sequence(
    start_date = "2017-01-01",
    end_date   = "2017-12-31",
    calendar   = "NYSE")

weekends <- tk_make_weekend_sequence(
    start_date = "2017-01-01",
    end_date   = "2017-12-31"
)

FANG %>%
    group_by(symbol) %>%
    future_frame(
        .length_out       = "1 year",
        .skip_values      = c(holidays, weekends)
    )
```

---

is_date_class *Check if an object is a date class*

Description

Check if an object is a date class

Usage

```
is_date_class(x)
```

Arguments

x	A vector to check

Value

Logical (TRUE/FALSE)

Examples

```
library(dplyr)

tk_make_timeseries("2011") %>% is_date_class()

letters %>% is_date_class()
```

---

lag_vec *Lag Transformation*

Description

`lag_vec()` applies a Lag Transformation.

Usage

```
lag_vec(x, lag = 1)

lead_vec(x, lag = -1)
```

Arguments

x	A vector to be lagged.
lag	Which lag (how far back) to be included in the differencing calculation. Negative lags are leads.

Details

Benefits:

This function is `NA` padded by default so it works well with `dplyr::mutate()` operations. The function allows both lags and leads (negative lags).

Lag Calculation

A lag is an offset of `lag` periods. `NA` values are returned for the number of `lag` periods.

Lead Calculation

A *negative lag* is considered a lead. The only difference between `lead_vec()` and `lag_vec()` is that the `lead_vec()` function contains a starting negative value.

Value

A numeric vector

See Also

Modeling and Advanced Lagging:

- `recipes::step_lag()` - Recipe for adding lags in `tidymodels` modeling
- `tk_augment_lags()` - Add many lags group-wise to a data.frame (tibble)

Vectorized Transformations:

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`, `ts_clean_vec()`

Examples

```
library(dplyr)

# --- VECTOR ----

# Lag
1:10 %>% lag_vec(lag = 1)

# Lead
1:10 %>% lag_vec(lag = -1)


# --- MUTATE ----

m4_daily %>%
    group_by(id) %>%
    mutate(lag_1 = lag_vec(value, lag = 1))
```

log_interval_vec *Log-Interval Transformation for Constrained Interval Forecasting*

Description

The `log_interval_vec()` transformation constrains a forecast to an interval specified by an `upper_limit` and a `lower_limit`. The transformation provides similar benefits to `log()` transformation, while ensuring the inverted transformation stays within an upper and lower limit.

Usage

```
log_interval_vec(
  x,
  limit_lower = "auto",
  limit_upper = "auto",
  offset = 0,
  silent = FALSE
)

log_interval_inv_vec(x, limit_lower, limit_upper, offset = 0)
```

Arguments

`x`	A positive numeric vector.
`limit_lower`	A lower limit. Must be less than the minimum value. If set to "auto", selects zero.
`limit_upper`	An upper limit. Must be greater than the maximum value. If set to "auto", selects a value that is 10% greater than the maximum value.
`offset`	An offset to include in the log transformation. Useful when the data contains values less than or equal to zero.
`silent`	Whether or not to report the parameter selections as a message.

Details

Log Interval Transformation

The Log Interval Transformation constrains values to specified upper and lower limits. The transformation maps limits to a function:

`log(((x + offset) - a)/(b - (x + offset)))`

where `a` is the lower limit and `b` is the upper limit

Inverse Transformation

The inverse transformation:

`(b-a)*(exp(x)) / (1 + exp(x)) + a - offset`

Value

A `numeric` vector of the transformed series.

References

- Forecasting: Principles & Practices: Forecasts constrained to an interval

See Also

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation & Anomaly Cleaning for Time Series: `ts_impute_vec()`, `ts_clean_vec()`

Other common transformations to reduce variance: `log()`, `log1p()` and `sqrt()`

Examples

```
library(dplyr)

values_trans <- log_interval_vec(1:10, limit_lower = 0, limit_upper = 11)
values_trans

values_trans_forecast <- c(values_trans, 3.4, 4.4, 5.4)

values_trans_forecast %>%
    log_interval_inv_vec(limit_lower = 0, limit_upper = 11) %>%
    plot()
```

`m4_daily` *Sample of 4 Daily Time Series Datasets from the M4 Competition*

Description

The fourth M Competition. M4, started on 1 January 2018 and ended in 31 May 2018. The competition included 100,000 time series datasets. This dataset includes **a sample of 4 daily time series from the competition.**

Usage

```
m4_daily
```

Format

A tibble: 9,743 x 3

- `id` Factor. Unique series identifier (4 total)
- `date` Date. Timestamp information. Daily format.
- `value` Numeric. Value at the corresponding timestamp.

Details

This is a sample of 4 daily data sets from the M4 competition.

Source

- M4 Competition Website

Examples

```
m4_daily
```

`m4_hourly` *Sample of 4 Hourly Time Series Datasets from the M4 Competition*

Description

The fourth M Competition. M4, started on 1 January 2018 and ended in 31 May 2018. The competition included 100,000 time series datasets. This dataset includes **a sample of 4 hourly time series from the competition.**

Usage

```
m4_hourly
```

Format

A tibble: 3,060 x 3

- `id` Factor. Unique series identifier (4 total)
- `date` Date-time. Timestamp information. Hourly format.
- `value` Numeric. Value at the corresponding timestamp.

Details

This is a sample of 4 hourly data sets from the M4 competition.

Source

- M4 Competition Website

Examples

```
m4_hourly
```

---

m4_monthly *Sample of 4 Monthly Time Series Datasets from the M4 Competition*

Description

The fourth M Competition. M4, started on 1 January 2018 and ended in 31 May 2018. The competition included 100,000 time series datasets. This dataset includes **a sample of 4 monthly time series from the competition.**

Usage

```
m4_monthly
```

Format

A tibble: 9,743 x 3

- `id` Factor. Unique series identifier (4 total)
- `date` Date. Timestamp information. Monthly format.
- `value` Numeric. Value at the corresponding timestamp.

Details

This is a sample of 4 Monthly data sets from the M4 competition.

Source

- M4 Competition Website

Examples

```
m4_monthly
```

m4_quarterly *Sample of 4 Quarterly Time Series Datasets from the M4 Competition*

Description

The fourth M Competition. M4, started on 1 January 2018 and ended in 31 May 2018. The competition included 100,000 time series datasets. This dataset includes **a sample of 4 quarterly time series from the competition.**

Usage

```
m4_quarterly
```

Format

A tibble: 9,743 x 3

- `id` Factor. Unique series identifier (4 total)
- `date` Date. Timestamp information. Quarterly format.
- `value` Numeric. Value at the corresponding timestamp.

Details

This is a sample of 4 Quarterly data sets from the M4 competition.

Source

- M4 Competition Website

Examples

```
m4_quarterly
```

m4_weekly *Sample of 4 Weekly Time Series Datasets from the M4 Competition*

Description

The fourth M Competition. M4, started on 1 January 2018 and ended in 31 May 2018. The competition included 100,000 time series datasets. This dataset includes **a sample of 4 weekly time series from the competition.**

Usage

```
m4_weekly
```

Format

A tibble: 9,743 x 3

- `id` Factor. Unique series identifier (4 total)
- `date` Date. Timestamp information. Weekly format.
- `value` Numeric. Value at the corresponding timestamp.

Details

This is a sample of 4 Weekly data sets from the M4 competition.

Source

- M4 Competition Website

Examples

```
m4_weekly
```

---

m4_yearly — *Sample of 4 Yearly Time Series Datasets from the M4 Competition*

Description

The fourth M Competition. M4, started on 1 January 2018 and ended in 31 May 2018. The competition included 100,000 time series datasets. This dataset includes **a sample of 4 yearly time series from the competition.**

Usage

```
m4_yearly
```

Format

A tibble: 9,743 x 3

- `id` Factor. Unique series identifier (4 total)
- `date` Date. Timestamp information. Yearly format.
- `value` Numeric. Value at the corresponding timestamp.

Details

This is a sample of 4 Yearly data sets from the M4 competition.

Source

- M4 Competition Website

Examples

```
m4_yearly
```

`mutate_by_time` *Mutate (for Time Series Data)*

Description

`mutate_by_time()` is a time-based variant of the popular `dplyr::mutate()` function that uses `.date_var` to specify a date or date-time column and `.by` to group the calculation by groups like "5 seconds", "week", or "3 months".

Usage

```
mutate_by_time(
  .data,
  .date_var,
  .by = "day",
  ...,
  .type = c("floor", "ceiling", "round")
)
```

Arguments

`.data` A `tbl` object or `data.frame`

`.date_var` A column containing date or date-time values to summarize. If missing, attempts to auto-detect date column.

`.by` A time unit to summarise by. Time units are collapsed using `lubridate::floor_date()` or `lubridate::ceiling_date()`.

The value can be:

- `second`
- `minute`
- `hour`
- `day`
- `week`
- `month`
- `bimonth`
- `quarter`
- `season`
- `halfyear`
- `year`

Arbitrary unique English abbreviations as in the `lubridate::period()` constructor are allowed.

`...` Name-value pairs. The name gives the name of the column in the output.

The value can be:

- A vector of length 1, which will be recycled to the correct length.
- A vector the same length as the current group (or the whole data frame if ungrouped).
- `NULL`, to remove the column.
- A data frame or tibble, to create multiple columns in the output.

`.type` One of "floor", "ceiling", or "round. Defaults to "floor". See `lubridate::round_date`.

Value

A `tibble` or `data.frame`

See Also

Time-Based dplyr functions:

- `summarise_by_time()` - Easily summarise using a date column.
- `mutate_by_time()` - Simplifies applying mutations by time windows.
- `pad_by_time()` - Insert time series rows with regularly spaced timestamps
- `filter_by_time()` - Quickly filter using date ranges.
- `filter_period()` - Apply filtering expressions inside periods (windows)
- `slice_period()` - Apply slice inside periods (windows)
- `condense_period()` - Convert to a different periodicity
- `between_time()` - Range detection for date or date-time sequences.
- `slidify()` - Turn any function into a sliding (rolling) function

Examples

```
# Libraries
library(dplyr)

# First value in each month
m4_daily_first_by_month_tbl <- m4_daily %>%
    group_by(id) %>%
    mutate_by_time(
        .date_var = date,
        .by       = "month", # Setup for monthly aggregation
        # mutate recycles a single value
        first_value_by_month  = first(value)
    )
m4_daily_first_by_month_tbl

# Visualize Time Series vs 1st Value Each Month
m4_daily_first_by_month_tbl %>%
    tidyr::pivot_longer(value:first_value_by_month) %>%
    plot_time_series(date, value, name,
```

```
.facet_scale = "free", .facet_ncol = 2,
.smooth = FALSE, .interactive = FALSE)
```

---

normalize_vec *Normalize to Range (0, 1)*

Description

Normalization is commonly used to center and scale numeric features to prevent one from dominating in algorithms that require data to be on the same scale.

Usage

```
normalize_vec(x, min = NULL, max = NULL, silent = FALSE)

normalize_inv_vec(x, min, max)
```

Arguments

x	A numeric vector.
min	The population min value in the normalization process.
max	The population max value in the normalization process.
silent	Whether or not to report the automated `min` and `max` parameters as a message.

Details

Standardization vs Normalization

- **Standardization** refers to a transformation that reduces the range to mean 0, standard deviation 1
- **Normalization** refers to a transformation that reduces the min-max range: (0, 1)

Value

A `numeric` vector with the transformation applied.

See Also

- Normalization/Standardization: `standardize_vec()`, `normalize_vec()`
- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`, `ts_clean_vec()`

Examples

```
library(dplyr)

d10_daily <- m4_daily %>% dplyr::filter(id == "D10")

# --- VECTOR ----

value_norm <- normalize_vec(d10_daily$value)
value      <- normalize_inv_vec(value_norm,
                                min = 1781.6,
                                max = 2649.3)

# --- MUTATE ----

m4_daily %>%
    group_by(id) %>%
    mutate(value_norm = normalize_vec(value))
```

`pad_by_time` *Insert time series rows with regularly spaced timestamps*

Description

The easiest way to fill in missing timestamps or convert to a more granular period (e.g. quarter to month). Wraps the `padr::pad()` function for padding tibbles.

Usage

```
pad_by_time(
  .data,
  .date_var,
  .by = "auto",
  .pad_value = NA,
  .fill_na_direction = c("none", "down", "up", "downup", "updown"),
  .start_date = NULL,
  .end_date = NULL
)
```

Arguments

`.data`	A tibble with a time-based column.
`.date_var`	A column containing date or date-time values to pad
`.by`	Either "auto", a time-based frequency like "year", "month", "day", "hour", etc, or a time expression like "5 min", or "7 days". See Details.
`.pad_value`	Fills in padded values. Default is `NA`.

.fill_na_direction
Users can provide an NA fill strategy using tidyr::fill(). Possible values: 'none', 'down', 'up', 'downup', 'updown'. Default: 'none'

.start_date
Specifies the start of the padded series. If NULL it will use the lowest value of the input variable.

.end_date
Specifies the end of the padded series. If NULL it will use the highest value of the input variable.

Details

Padding Missing Observations

The most common use case for pad_by_time() is to add rows where timestamps are missing. This could be from sales data that have missing values on weekends and holidays. Or it could be high frequency data where observations are irregularly spaced and should be reset to a regular frequency.

Going from Low to High Frequency

The second use case is going from a low frequency (e.g. day) to high frequency (e.g. hour). This is possible by supplying a higher frequency to pad_by_time().

Interval, .by

Padding can be applied in the following ways:

- .by = "auto" - pad_by_time() will detect the time-stamp frequency and apply padding.
- The eight intervals in are: year, quarter, month, week, day, hour, min, and sec.
- Intervals like 5 minutes, 6 hours, 10 days are possible.

Pad Value, .pad_value

A pad value can be supplied that fills in missing numeric data. Note that this is only applied to numeric columns.

Fill NA Direction, .fill_na_directions

Uses tidyr::fill() to fill missing observations using a fill strategy.

Value

A tibble or data.frame with rows containing missing timestamps added.

References

- This function wraps the padr::pad() function developed by Edwin Thoen.

See Also

Imputation:

- ts_impute_vec() - Impute missing values for time series.

Time-Based dplyr functions:

- summarise_by_time() - Easily summarise using a date column.

- mutate_by_time() - Simplifies applying mutations by time windows.
- pad_by_time() - Insert time series rows with regularly spaced timestamps
- filter_by_time() - Quickly filter using date ranges.
- filter_period() - Apply filtering expressions inside periods (windows)
- slice_period() - Apply slice inside periods (windows)
- condense_period() - Convert to a different periodicity
- between_time() - Range detection for date or date-time sequences.
- slidify() - Turn any function into a sliding (rolling) function

Examples

```
library(dplyr)

# Create a quarterly series with 1 missing value
missing_data_tbl <- tibble::tibble(
    date = tk_make_timeseries("2014-01-01", "2015-01-01", by = "quarter"),
    value = 1:5
) %>%
    slice(-4) # Lose the 4th quarter on purpose
missing_data_tbl


# Detects missing quarter, and pads the missing regularly spaced quarter with NA
missing_data_tbl %>% pad_by_time(date, .by = "quarter")

# Can specify a shorter period. This fills monthly.
missing_data_tbl %>% pad_by_time(date, .by = "month")

# Can let pad_by_time() auto-detect date and period
missing_data_tbl %>% pad_by_time()

# Can specify a .pad_value
missing_data_tbl %>% pad_by_time(date, .by = "quarter", .pad_value = 0)

# Can then impute missing values
missing_data_tbl %>%
    pad_by_time(date, .by = "quarter") %>%
    mutate(value = ts_impute_vec(value, period = 1))

# Can specify a custom .start_date and .end_date
missing_data_tbl %>%
   pad_by_time(date, .by = "quarter", .start_date = "2013", .end_date = "2015-07-01")

# Can specify a tidyr::fill() direction
missing_data_tbl %>%
   pad_by_time(date, .by = "quarter",
               .fill_na_direction = "downup",
               .start_date = "2013", .end_date = "2015-07-01")

# --- GROUPS ----
```

```
# Apply standard NA padding to groups
FANG %>%
    group_by(symbol) %>%
    pad_by_time(.by = "day")

# Apply constant pad value
FANG %>%
    group_by(symbol) %>%
    pad_by_time(.by = "day", .pad_value = 0)

# Apply filled padding to groups
FANG %>%
    group_by(symbol) %>%
    pad_by_time(.by = "day", .fill_na_direction = "down")
```

---

parse_date2 *Fast, flexible date and datetime parsing*

Description

Significantly faster time series parsing than `readr::parse_date`, `readr::parse_datetime`, `lubridate::as_date()`, and `lubridate::as_datetime()`. Uses `anytime` package, which relies on `Boost.Date_Time` C++ library for date/datetime parsing.

Usage

```
parse_date2(x, ..., silent = FALSE)

parse_datetime2(x, tz = "UTC", tz_shift = FALSE, ..., silent = FALSE)
```

Arguments

x	A character vector
...	Additional parameters passed to `anytime()` and `anydate()`
silent	If `TRUE`, warns the user of parsing failures.
tz	Datetime only. A timezone (see `OlsenNames()`).
tz_shift	Datetime only. If FALSE, forces the datetime into the time zone. If TRUE, offsets the datetime from UTC to the new time zone.

Details

Parsing Formats

- Date Formats: Must follow a Year, Month, Day sequence. (e.g. `parse_date2("2011 June")` is OK, `parse_date2("June 2011")` is NOT OK).

- Date Time Formats: Must follow a YMD HMS sequence.

Refer to `lubridate::mdy()` for Month, Day, Year and additional formats.

Time zones (Datetime)

Time zones are handled in a similar way to `lubridate::as_datetime()` in that time zones are forced rather than shifted. This is a key difference between `anytime::anytime()`, which shifts datetimes to the specified timezone by default.

Value

Returns a `date` or `datatime` vector from the transformation applied to character timestamp vector.

References

- This function wraps the `anytime::anytime()` and `anytime::anydate()` functions developed by Dirk Eddelbuettel.

Examples

```
# Fast date parsing
parse_date2("2011")
parse_date2("2011 June 3rd")

# Fast datetime parsing
parse_datetime2("2011")
parse_datetime2("2011 Jan 1 12:35:21")

# Time Zones (datetime only)
parse_datetime2("2011 Jan 1 12:35:21", tz = "Europe/London")
```

`plot_acf_diagnostics` *Visualize the ACF, PACF, and CCFs for One or More Time Series*

Description

Returns the **ACF and PACF of a target** and optionally **CCF's of one or more lagged predictors** in interactive `plotly` plots. Scales to multiple time series with `group_by()`.

Usage

```
plot_acf_diagnostics(
  .data,
  .date_var,
  .value,
  .ccf_vars = NULL,
  .lags = 1000,
  .show_ccf_vars_only = FALSE,
```

```
  .show_white_noise_bars = TRUE,
  .facet_ncol = 1,
  .facet_scales = "fixed",
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_alpha = 1,
  .point_color = "#2c3e50",
  .point_size = 1,
  .point_alpha = 1,
  .x_intercept = NULL,
  .x_intercept_color = "#E31A1C",
  .hline_color = "#2c3e50",
  .white_noise_line_type = 2,
  .white_noise_line_color = "#A6CEE3",
  .title = "Lag Diagnostics",
  .x_lab = "Lag",
  .y_lab = "Correlation",
  .interactive = TRUE,
  .plotly_slider = FALSE
)
```

Arguments

`.data`	A data frame or tibble with numeric features (values) in descending chronological order
`.date_var`	A column containing either date or date-time values
`.value`	A numeric column with a value to have ACF and PACF calculations performed.
`.ccf_vars`	Additional features to perform Lag Cross Correlations (CCFs) versus the `.value`. Useful for evaluating external lagged regressors.
`.lags`	A sequence of one or more lags to evaluate.
`.show_ccf_vars_only`	Hides the ACF and PACF plots so you can focus on only CCFs.
`.show_white_noise_bars`	Shows the white noise significance bounds.
`.facet_ncol`	Facets: Number of facet columns. Has no effect if using `grouped_df`.
`.facet_scales`	Facets: Options include "fixed", "free", "free_y", "free_x"
`.line_color`	Line color. Use keyword: "scale_color" to change the color by the facet.
`.line_size`	Line size (linewidth)
`.line_alpha`	Line opacity. Adjust the transparency of the line. Range: (0, 1)
`.point_color`	Point color. Use keyword: "scale_color" to change the color by the facet.
`.point_size`	Point size
`.point_alpha`	Opacity. Adjust the transparency of the points. Range: (0, 1)
`.x_intercept`	Numeric lag. Adds a vertical line.
`.x_intercept_color`	Color for the x-intercept line.

`.hline_color` Color for the y-intercept = 0 line.

`.white_noise_line_type`
Line type for white noise bars. Set to 2 for "dashed" by default.

`.white_noise_line_color`
Line color for white noise bars. Set to `tidyquant::palette_light()` "steel blue" by default.

`.title` Title for the plot

`.x_lab` X-axis label for the plot

`.y_lab` Y-axis label for the plot

`.interactive` Returns either a static (`ggplot2`) visualization or an interactive (`plotly`) visualization

`.plotly_slider` If TRUE, returns a plotly x-axis range slider.

Details

Simplified ACF, PACF, & CCF

We are often interested in all 3 of these functions. Why not get all 3+ at once? Now you can.

- **ACF** - Autocorrelation between a target variable and lagged versions of itself
- **PACF** - Partial Autocorrelation removes the dependence of lags on other lags highlighting key seasonalities.
- **CCF** - Shows how lagged predictors can be used for prediction of a target variable.

Lag Specification

Lags (`.lags`) can either be specified as:

- A time-based phrase indicating a duraction (e.g. `2 months`)
- A maximum lag (e.g. `.lags = 28`)
- A sequence of lags (e.g. `.lags = 7:28`)

Scales to Multiple Time Series with Groups

The `plot_acf_diagnostics()` works with `grouped_df`'s, meaning you can group your time series by one or more categorical columns with `dplyr::group_by()` and then apply `plot_acf_diagnostics()` to return group-wise lag diagnostics.

Special Note on Groups

Unlike other plotting utilities, the `.facet_vars` arguments is NOT included. Use `dplyr::group_by()` for processing multiple time series groups.

Calculating the White Noise Significance Bars

The formula for the significance bars is `+2/sqrt(T)` and `-2/sqrt(T)` where `T` is the length of the time series. For a white noise time series, 95% of the data points should fall within this range. Those that don't may be significant autocorrelations.

Value

A static `ggplot2` plot or an interactive `plotly` plot

See Also

- **Visualizing ACF, PACF, & CCF:** plot_acf_diagnostics()
- **Visualizing Seasonality:** plot_seasonal_diagnostics()
- **Visualizing Time Series:** plot_time_series()

Examples

```
library(dplyr)
library(ggplot2)


# Apply Transformations
# - Differencing transformation to identify ARIMA & SARIMA Orders
m4_hourly %>%
    group_by(id) %>%
    plot_acf_diagnostics(
        date, value,               # ACF & PACF
        .lags = "7 days",          # 7-Days of hourly lags
        .interactive = FALSE
    )

# Apply Transformations
# - Differencing transformation to identify ARIMA & SARIMA Orders
m4_hourly %>%
    group_by(id) %>%
    plot_acf_diagnostics(
        date,
        diff_vec(value, lag = 1), # Difference the value column
        .lags        = 0:(24*7),   # 7-Days of hourly lags
        .interactive = FALSE
    ) +
    ggtitle("ACF Diagnostics",  subtitle = "1st Difference")

# CCFs Too!
walmart_sales_weekly %>%
    select(id, Date, Weekly_Sales, Temperature, Fuel_Price) %>%
    group_by(id) %>%
    plot_acf_diagnostics(
        Date, Weekly_Sales,                        # ACF & PACF
        .ccf_vars    = c(Temperature, Fuel_Price), # CCFs
        .lags        = "3 months", # 3 months of weekly lags
        .interactive = FALSE
    )
```

---

plot_anomalies *Visualize Anomalies for One or More Time Series*

Description

`plot_anomalies()` is an interactive and scalable function for visualizing anomalies in time series data. Plots are available in interactive `plotly` (default) and static `ggplot2` format.

`plot_anomalies_decomp()`: Takes in data from the `anomalize()` function, and returns a plot of the anomaly decomposition. Useful for interpeting how the `anomalize()` function is determining outliers from "remainder".

`plot_anomalies_cleaned()` helps users visualize the before/after of cleaning anomalies.

Usage

```
plot_anomalies(
  .data,
  .date_var,
  .facet_vars = NULL,
  .facet_ncol = 1,
  .facet_nrow = 1,
  .facet_scales = "free",
  .facet_dir = "h",
  .facet_collapse = FALSE,
  .facet_collapse_sep = " ",
  .facet_strip_remove = FALSE,
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
  .line_alpha = 1,
  .anom_color = "#e31a1c",
  .anom_alpha = 1,
  .anom_size = 1.5,
  .ribbon_fill = "grey20",
  .ribbon_alpha = 0.2,
  .legend_show = TRUE,
  .title = "Anomaly Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Anomaly",
  .interactive = TRUE,
  .trelliscope = FALSE,
  .trelliscope_params = list()
)

plot_anomalies_decomp(
  .data,
  .date_var,
  .facet_vars = NULL,
  .facet_scales = "free",
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
```

```
  .line_alpha = 1,
  .title = "Anomaly Decomposition Plot",
  .x_lab = "",
  .y_lab = "",
  .interactive = TRUE
)

plot_anomalies_cleaned(
  .data,
  .date_var,
  .facet_vars = NULL,
  .facet_ncol = 1,
  .facet_nrow = 1,
  .facet_scales = "free",
  .facet_dir = "h",
  .facet_collapse = FALSE,
  .facet_collapse_sep = " ",
  .facet_strip_remove = FALSE,
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
  .line_alpha = 1,
  .cleaned_line_color = "#e31a1c",
  .cleaned_line_size = 0.5,
  .cleaned_line_type = 1,
  .cleaned_line_alpha = 1,
  .legend_show = TRUE,
  .title = "Anomalies Cleaned Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Legend",
  .interactive = TRUE,
  .trelliscope = FALSE,
  .trelliscope_params = list()
)
```

Arguments

`.data`	A `tibble` or `data.frame` that has been anomalized by `anomalize()`
`.date_var`	A column containing either date or date-time values
`.facet_vars`	One or more grouping columns that broken out into `ggplot2` facets. These can be selected using `tidyselect()` helpers (e.g `contains()`).
`.facet_ncol`	Number of facet columns.
`.facet_nrow`	Number of facet rows (only used for `.trelliscope = TRUE`)
`.facet_scales`	Control facet x & y-axis ranges. Options include "fixed", "free", "free_y", "free_x"
`.facet_dir`	The direction of faceting ("h" for horizontal, "v" for vertical). Default is "h".

.facet_collapse
: Multiple facets included on one facet strip instead of multiple facet strips.

.facet_collapse_sep
: The separator used for collapsing facets.

.facet_strip_remove
: Whether or not to remove the strip and text label for each facet.

.line_color
: Line color.

.line_size
: Line size.

.line_type
: Line type.

.line_alpha
: Line alpha (opacity). Range: (0, 1).

.anom_color
: Color for the anomaly dots

.anom_alpha
: Opacity for the anomaly dots. Range: (0, 1).

.anom_size
: Size for the anomaly dots

.ribbon_fill
: Fill color for the acceptable range

.ribbon_alpha
: Fill opacity for the acceptable range. Range: (0, 1).

.legend_show
: Toggles on/off the Legend

.title
: Plot title.

.x_lab
: Plot x-axis label

.y_lab
: Plot y-axis label

.color_lab
: Plot label for the color legend

.interactive
: If TRUE, returns a `plotly` interactive plot. If FALSE, returns a static `ggplot2` plot.

.trelliscope
: Returns either a normal plot or a trelliscopejs plot (great for many time series) Must have `trelliscopejs` installed.

.trelliscope_params
: Pass parameters to the `trelliscopejs::facet_trelliscope()` function as a `list()`. The only parameters that cannot be passed are:
  - `ncol`: use `.facet_ncol`
  - `nrow`: use `.facet_nrow`
  - `scales`: use `facet_scales`
  - `as_plotly`: use `.interactive`

.cleaned_line_color
: Line color.

.cleaned_line_size
: Line size.

.cleaned_line_type
: Line type.

.cleaned_line_alpha
: Line alpha (opacity). Range: (0, 1).

Value

A `plotly` or `ggplot2` visualization

Examples

```
# Plot Anomalies
library(dplyr)

walmart_sales_weekly %>%
    filter(id %in% c("1_1", "1_3")) %>%
    group_by(id) %>%
    anomalize(Date, Weekly_Sales) %>%
    plot_anomalies(Date, .facet_ncol = 2, .ribbon_alpha = 0.25, .interactive = FALSE)

# Plot Anomalies Decomposition
library(dplyr)

walmart_sales_weekly %>%
    filter(id %in% c("1_1", "1_3")) %>%
    group_by(id) %>%
    anomalize(Date, Weekly_Sales, .message = FALSE) %>%
    plot_anomalies_decomp(Date, .interactive = FALSE)

# Plot Anomalies Cleaned
library(dplyr)

walmart_sales_weekly %>%
    filter(id %in% c("1_1", "1_3")) %>%
    group_by(id) %>%
    anomalize(Date, Weekly_Sales, .message = FALSE) %>%
    plot_anomalies_cleaned(Date, .facet_ncol = 2, .interactive = FALSE)
```

`plot_anomaly_diagnostics` *Visualize Anomalies for One or More Time Series*

Description

An interactive and scalable function for visualizing anomalies in time series data. Plots are available in interactive `plotly` (default) and static `ggplot2` format.

Usage

```
plot_anomaly_diagnostics(
  .data,
  .date_var,
  .value,
  .facet_vars = NULL,
  .frequency = "auto",
  .trend = "auto",
  .alpha = 0.05,
  .max_anomalies = 0.2,
```

```
  .message = TRUE,
  .facet_ncol = 1,
  .facet_nrow = 1,
  .facet_scales = "free",
  .facet_dir = "h",
  .facet_collapse = FALSE,
  .facet_collapse_sep = " ",
  .facet_strip_remove = FALSE,
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
  .line_alpha = 1,
  .anom_color = "#e31a1c",
  .anom_alpha = 1,
  .anom_size = 1.5,
  .ribbon_fill = "grey20",
  .ribbon_alpha = 0.2,
  .legend_show = TRUE,
  .title = "Anomaly Diagnostics",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Anomaly",
  .interactive = TRUE,
  .trelliscope = FALSE,
  .trelliscope_params = list()
)
```

Arguments

`.data`	A `tibble` or `data.frame` with a time-based column
`.date_var`	A column containing either date or date-time values
`.value`	A column containing numeric values
`.facet_vars`	One or more grouping columns that broken out into `ggplot2` facets. These can be selected using `tidyselect()` helpers (e.g `contains()`).
`.frequency`	Controls the seasonal adjustment (removal of seasonality). Input can be either "auto", a time-based definition (e.g. "2 weeks"), or a numeric number of observations per frequency (e.g. 10). Refer to `tk_get_frequency()`.
`.trend`	Controls the trend component. For STL, trend controls the sensitivity of the LOESS smoother, which is used to remove the remainder. Refer to `tk_get_trend()`.
`.alpha`	Controls the width of the "normal" range. Lower values are more conservative while higher values are less prone to incorrectly classifying "normal" observations.
`.max_anomalies`	The maximum percent of anomalies permitted to be identified.
`.message`	A boolean. If `TRUE`, will output information related to automatic frequency and trend selection (if applicable).
`.facet_ncol`	Number of facet columns.

`.facet_nrow` Number of facet rows (only used for `.trelliscope = TRUE`)

`.facet_scales` Control facet x & y-axis ranges. Options include "fixed", "free", "free_y", "free_x"

`.facet_dir` The direction of faceting ("h" for horizontal, "v" for vertical). Default is "h".

`.facet_collapse` Multiple facets included on one facet strip instead of multiple facet strips.

`.facet_collapse_sep` The separator used for collapsing facets.

`.facet_strip_remove` Whether or not to remove the strip and text label for each facet.

`.line_color` Line color.

`.line_size` Line size.

`.line_type` Line type.

`.line_alpha` Line alpha (opacity). Range: (0, 1).

`.anom_color` Color for the anomaly dots

`.anom_alpha` Opacity for the anomaly dots. Range: (0, 1).

`.anom_size` Size for the anomaly dots

`.ribbon_fill` Fill color for the acceptable range

`.ribbon_alpha` Fill opacity for the acceptable range. Range: (0, 1).

`.legend_show` Toggles on/off the Legend

`.title` Plot title.

`.x_lab` Plot x-axis label

`.y_lab` Plot y-axis label

`.color_lab` Plot label for the color legend

`.interactive` If TRUE, returns a `plotly` interactive plot. If FALSE, returns a static `ggplot2` plot.

`.trelliscope` Returns either a normal plot or a trelliscopejs plot (great for many time series) Must have `trelliscopejs` installed.

`.trelliscope_params` Pass parameters to the `trelliscopejs::facet_trelliscope()` function as a `list()`. The only parameters that cannot be passed are:

- `ncol`: use `.facet_ncol`
- `nrow`: use `.facet_nrow`
- `scales`: use `facet_scales`
- `as_plotly`: use `.interactive`

Details

The `plot_anomaly_diagnostics()` is a visualization wrapper for `tk_anomaly_diagnostics()` group-wise anomaly detection, implements a 2-step process to detect outliers in time series.

Step 1: Detrend & Remove Seasonality using STL Decomposition

The decomposition separates the "season" and "trend" components from the "observed" values leaving the "remainder" for anomaly detection.

The user can control two parameters: frequency and trend.

1. `.frequency`: Adjusts the "season" component that is removed from the "observed" values.
2. `.trend`: Adjusts the trend window (t.window parameter from `stats::stl()` that is used.

The user may supply both `.frequency` and `.trend` as time-based durations (e.g. "6 weeks") or numeric values (e.g. 180) or "auto", which predetermines the frequency and/or trend based on the scale of the time series using the `tk_time_scale_template()`.

Step 2: Anomaly Detection

Once "trend" and "season" (seasonality) is removed, anomaly detection is performed on the "remainder". Anomalies are identified, and boundaries (recomposed_l1 and recomposed_l2) are determined.

The Anomaly Detection Method uses an inner quartile range (IQR) of +/-25 the median.

IQR Adjustment, alpha parameter

With the default `alpha = 0.05`, the limits are established by expanding the 25/75 baseline by an IQR Factor of 3 (3X). The *IQR Factor = 0.15 / alpha* (hence 3X with alpha = 0.05):

- To increase the IQR Factor controlling the limits, decrease the alpha, which makes it more difficult to be an outlier.
- Increase alpha to make it easier to be an outlier.
- The IQR outlier detection method is used in `forecast::tsoutliers()`.
- A similar outlier detection method is used by Twitter's `AnomalyDetection` package.
- Both Twitter and Forecast tsoutliers methods have been implemented in Business Science's `anomalize` package.

Value

A `plotly` or `ggplot2` visualization

References

1. CLEVELAND, R. B., CLEVELAND, W. S., MCRAE, J. E., AND TERPENNING, I. STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Journal of Official Statistics, Vol. 6, No. 1 (1990), pp. 3-73.
2. Owen S. Vallis, Jordan Hochenbaum and Arun Kejariwal (2014). A Novel Technique for Long-Term Anomaly Detection in the Cloud. Twitter Inc.

See Also

- `tk_anomaly_diagnostics()`: Group-wise anomaly detection

Examples

```
library(dplyr)

walmart_sales_weekly %>%
    group_by(id) %>%
    plot_anomaly_diagnostics(Date, Weekly_Sales,
                             .message = FALSE,
                             .facet_ncol = 3,
                             .ribbon_alpha = 0.25,
                             .interactive = FALSE)
```

---

plot_seasonal_diagnostics

Visualize Multiple Seasonality Features for One or More Time Series

---

Description

An interactive and scalable function for visualizing time series seasonality. Plots are available in interactive plotly (default) and static ggplot2 format.

Usage

```
plot_seasonal_diagnostics(
  .data,
  .date_var,
  .value,
  .facet_vars = NULL,
  .feature_set = "auto",
  .geom = c("boxplot", "violin"),
  .geom_color = "#2c3e50",
  .geom_outlier_color = "#2c3e50",
  .title = "Seasonal Diagnostics",
  .x_lab = "",
  .y_lab = "",
  .interactive = TRUE
)
```

Arguments

.data	A tibble or data.frame with a time-based column
.date_var	A column containing either date or date-time values
.value	A column containing numeric values
.facet_vars	One or more grouping columns that broken out into ggplot2 facets. These can be selected using tidyselect() helpers (e.g contains()).

`.feature_set` One or multiple selections to analyze for seasonality. Choices include:

- "auto" - Automatically selects features based on the time stamps and length of the series.
- "second" - Good for analyzing seasonality by second of each minute.
- "minute" - Good for analyzing seasonality by minute of the hour
- "hour" - Good for analyzing seasonality by hour of the day
- "wday.lbl" - Labeled weekdays. Good for analyzing seasonality by day of the week.
- "week" - Good for analyzing seasonality by week of the year.
- "month.lbl" - Labeled months. Good for analyzing seasonality by month of the year.
- "quarter" - Good for analyzing seasonality by quarter of the year
- "year" - Good for analyzing seasonality over multiple years.

`.geom` Either "boxplot" or "violin"

`.geom_color` Geometry color. Line color. Use keyword: "scale_color" to change the color by the facet.

`.geom_outlier_color`
Color used to highlight outliers.

`.title` Plot title.

`.x_lab` Plot x-axis label

`.y_lab` Plot y-axis label

`.interactive` If TRUE, returns a `plotly` interactive plot. If FALSE, returns a static `ggplot2` plot.

Details

Automatic Feature Selection

Internal calculations are performed to detect a sub-range of features to include useing the following logic:

- The *minimum* feature is selected based on the median difference between consecutive timestamps
- The *maximum* feature is selected based on having 2 full periods.

Example: Hourly timestamp data that lasts more than 2 weeks will have the following features: "hour", "wday.lbl", and "week".

Scalable with Grouped Data Frames

This function respects grouped `data.frame` and `tibbles` that were made with `dplyr::group_by()`.

For grouped data, the automatic feature selection returned is a collection of all features within the sub-groups. This means extra features are returned even though they may be meaningless for some of the groups.

Transformations

The `.value` parameter respects transformations (e.g. `.value = log(sales)`).

Value

A plotly or ggplot2 visualization

Examples

```
library(dplyr)

# ---- MULTIPLE FREQUENCY ----
# Taylor 30-minute dataset from forecast package
taylor_30_min

# Visualize series
taylor_30_min %>%
    plot_time_series(date, value, .interactive = FALSE)

# Visualize seasonality
taylor_30_min %>%
    plot_seasonal_diagnostics(date, value, .interactive = FALSE)

# ---- GROUPED EXAMPLES ----
# m4 hourly dataset
m4_hourly

# Visualize series
m4_hourly %>%
    group_by(id) %>%
    plot_time_series(date, value, .facet_scales = "free", .interactive = FALSE)

# Visualize seasonality
m4_hourly %>%
    group_by(id) %>%
    plot_seasonal_diagnostics(date, value, .interactive = FALSE)
```

---

plot_stl_diagnostics *Visualize STL Decomposition Features for One or More Time Series*

Description

An interactive and scalable function for visualizing time series STL Decomposition. Plots are available in interactive plotly (default) and static ggplot2 format.

Usage

```
plot_stl_diagnostics(
  .data,
  .date_var,
```

```
  .value,
  .facet_vars = NULL,
  .feature_set = c("observed", "season", "trend", "remainder", "seasadj"),
  .frequency = "auto",
  .trend = "auto",
  .message = TRUE,
  .facet_scales = "free",
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
  .line_alpha = 1,
  .title = "STL Diagnostics",
  .x_lab = "",
  .y_lab = "",
  .interactive = TRUE
)
```

Arguments

`.data`	A `tibble` or `data.frame` with a time-based column
`.date_var`	A column containing either date or date-time values
`.value`	A column containing numeric values
`.facet_vars`	One or more grouping columns that broken out into `ggplot2` facets. These can be selected using `tidyselect()` helpers (e.g `contains()`).
`.feature_set`	The STL decompositions to visualize. Select one or more of "observed", "season", "trend", "remainder", "seasadj".
`.frequency`	Controls the seasonal adjustment (removal of seasonality). Input can be either "auto", a time-based definition (e.g. "2 weeks"), or a numeric number of observations per frequency (e.g. 10). Refer to `tk_get_frequency()`.
`.trend`	Controls the trend component. For STL, trend controls the sensitivity of the lowess smoother, which is used to remove the remainder.
`.message`	A boolean. If `TRUE`, will output information related to automatic frequency and trend selection (if applicable).
`.facet_scales`	Control facet x & y-axis ranges. Options include "fixed", "free", "free_y", "free_x"
`.line_color`	Line color.
`.line_size`	Line size.
`.line_type`	Line type.
`.line_alpha`	Line alpha (opacity). Range: (0, 1).
`.title`	Plot title.
`.x_lab`	Plot x-axis label
`.y_lab`	Plot y-axis label
`.interactive`	If TRUE, returns a `plotly` interactive plot. If FALSE, returns a static `ggplot2` plot.

Details

The `plot_stl_diagnostics()` function generates a Seasonal-Trend-Loess decomposition. The function is "tidy" in the sense that it works on data frames and is designed to work with `dplyr` groups.

STL method:

The STL method implements time series decomposition using the underlying `stats::stl()`. The decomposition separates the "season" and "trend" components from the "observed" values leaving the "remainder".

Frequency & Trend Selection

The user can control two parameters: `.frequency` and `.trend`.

1. The `.frequency` parameter adjusts the "season" component that is removed from the "observed" values.
2. The `.trend` parameter adjusts the trend window (`t.window` parameter from `stl()`) that is used.

The user may supply both `.frequency` and `.trend` as time-based durations (e.g. "6 weeks") or numeric values (e.g. 180) or "auto", which automatically selects the frequency and/or trend based on the scale of the time series.

Value

A `plotly` or `ggplot2` visualization

Examples

```
library(dplyr)

# ---- SINGLE TIME SERIES DECOMPOSITION ----
m4_hourly %>%
    filter(id == "H10") %>%
    plot_stl_diagnostics(
        date, value,
        # Set features to return, desired frequency and trend
        .feature_set = c("observed", "season", "trend", "remainder"),
        .frequency   = "24 hours",
        .trend       = "1 week",
        .interactive = FALSE)


# ---- GROUPS ----
m4_hourly %>%
    group_by(id) %>%
    plot_stl_diagnostics(
        date, value,
        .feature_set = c("observed", "season", "trend"),
        .interactive = FALSE)
```

plot_time_series *Interactive Plotting for One or More Time Series*

Description

A workhorse time-series plotting function that generates interactive `plotly` plots, consolidates 20+ lines of `ggplot2` code, and scales well to many time series.

Usage

```
plot_time_series(
  .data,
  .date_var,
  .value,
  .color_var = NULL,
  .facet_vars = NULL,
  .facet_ncol = 1,
  .facet_nrow = 1,
  .facet_scales = "free_y",
  .facet_dir = "h",
  .facet_collapse = FALSE,
  .facet_collapse_sep = " ",
  .facet_strip_remove = FALSE,
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
  .line_alpha = 1,
  .y_intercept = NULL,
  .y_intercept_color = "#2c3e50",
  .x_intercept = NULL,
  .x_intercept_color = "#2c3e50",
  .smooth = TRUE,
  .smooth_period = "auto",
  .smooth_message = FALSE,
  .smooth_span = NULL,
  .smooth_degree = 2,
  .smooth_color = "#3366FF",
  .smooth_size = 1,
  .smooth_alpha = 1,
  .legend_show = TRUE,
  .title = "Time Series Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Legend",
  .interactive = TRUE,
  .plotly_slider = FALSE,
  .trelliscope = FALSE,
```

```
  .trelliscope_params = list()
)
```

Arguments

.data — A tibble or data.frame with a time-based column

.date_var — A column containing either date or date-time values

.value — A column containing numeric values

.color_var — A categorical column that can be used to change the line color

.facet_vars — One or more grouping columns that broken out into ggplot2 facets. These can be selected using tidyselect() helpers (e.g contains()).

.facet_ncol — Number of facet columns.

.facet_nrow — Number of facet rows (only used for .trelliscope = TRUE)

.facet_scales — Control facet x & y-axis ranges. Options include "fixed", "free", "free_y", "free_x"

.facet_dir — The direction of faceting ("h" for horizontal, "v" for vertical). Default is "h".

.facet_collapse — Multiple facets included on one facet strip instead of multiple facet strips.

.facet_collapse_sep — The separator used for collapsing facets.

.facet_strip_remove — Whether or not to remove the strip and text label for each facet.

.line_color — Line color. Overrided if .color_var is specified.

.line_size — Line size.

.line_type — Line type.

.line_alpha — Line alpha (opacity). Range: (0, 1).

.y_intercept — Value for a y-intercept on the plot

.y_intercept_color — Color for the y-intercept

.x_intercept — Value for a x-intercept on the plot

.x_intercept_color — Color for the x-intercept

.smooth — Logical - Whether or not to include a trendline smoother. Uses See smooth_vec() to apply a LOESS smoother.

.smooth_period — Number of observations to include in the Loess Smoother. Set to "auto" by default, which uses tk_get_trend() to determine a logical trend cycle.

.smooth_message — Logical. Whether or not to return the trend selected as a message. Useful for those that want to see what .smooth_period was selected.

.smooth_span — Percentage of observations to include in the Loess Smoother. You can use either period or span. See smooth_vec().

.smooth_degree Flexibility of Loess Polynomial. Either 0, 1, 2 (0 = lest flexible, 2 = more flexible).

.smooth_color Smoother line color

.smooth_size Smoother line size

.smooth_alpha Smoother alpha (opacity). Range: (0, 1).

.legend_show Toggles on/off the Legend

.title Title for the plot

.x_lab X-axis label for the plot

.y_lab Y-axis label for the plot

.color_lab Legend label if a color_var is used.

.interactive Returns either a static (ggplot2) visualization or an interactive (plotly) visualization

.plotly_slider If TRUE, returns a plotly date range slider.

.trelliscope Returns either a normal plot or a trelliscopejs plot (great for many time series) Must have trelliscopejs installed.

.trelliscope_params

Pass parameters to the trelliscopejs::facet_trelliscope() function as a list(). The only parameters that cannot be passed are:

- ncol: use .facet_ncol
- nrow: use .facet_nrow
- scales: use facet_scales
- as_plotly: use .interactive

Details

plot_time_series() is a scalable function that works with both *ungrouped* and *grouped* data.frame objects (and tibbles!).

Interactive by Default

plot_time_series() is built for exploration using:

- **Interactive Plots:** plotly (default) - Great for exploring!
- **Static Plots:** ggplot2 (set .interactive = FALSE) - Great for PDF Reports

By default, an interactive plotly visualization is returned.

Scalable with Facets & Dplyr Groups

plot_time_series() returns multiple time series plots using ggplot2 facets:

- group_by() - If groups are detected, multiple facets are returned
- plot_time_series(.facet_vars) - You can manually supply facets as well.

Can Transform Values just like ggplot

The .values argument accepts transformations just like ggplot2. For example, if you want to take the log of sales you can use a call like plot_time_series(date, log(sales)) and the log transformation will be applied.

Smoother Period / Span Calculation

The `.smooth = TRUE` option returns a smoother that is calculated based on either:

1. A `.smooth_period`: Number of observations
2. A `.smooth_span`: A percentage of observations

By default, the `.smooth_period` is automatically calculated using 75% of the observertions. This is the same as `geom_smooth(method = "loess", span = 0.75)`.

A user can specify a time-based window (e.g. `.smooth_period = "1 year"`) or a numeric value (e.g. `smooth_period = 365`).

Time-based windows return the median number of observations in a window using `tk_get_trend()`.

Value

A static `ggplot2` plot or an interactive `plotly` plot

Examples

```
library(dplyr)
library(lubridate)

# Works with individual time series
FANG %>%
    filter(symbol == "FB") %>%
    plot_time_series(date, adjusted, .interactive = FALSE)

# Works with groups
FANG %>%
    group_by(symbol) %>%
    plot_time_series(date, adjusted,
                     .facet_ncol  = 2,     # 2-column layout
                     .interactive = FALSE)

# Can also group inside & use .color_var
FANG %>%
    mutate(year = year(date)) %>%
    plot_time_series(date, adjusted,
                     .facet_vars     = c(symbol, year), # add groups/facets
                     .color_var      = year,            # color by year
                     .facet_ncol     = 4,
                     .facet_scales   = "free",
                     .facet_collapse = TRUE,  # combine group strip text into 1 line
                     .interactive    = FALSE)

# Can apply transformations to .value or .color_var
# - .value = log(adjusted)
# - .color_var = year(date)
FANG %>%
    plot_time_series(date, log(adjusted),
                     .color_var    = year(date),
                     .facet_vars   = contains("symbol"),
```

```
                      .facet_ncol   = 2,
                      .facet_scales = "free",
                      .y_lab        = "Log Scale",
                      .interactive  = FALSE)
```

---

plot_time_series_boxplot
Interactive Time Series Box Plots

Description

A boxplot function that generates interactive `plotly` plots for time series.

Usage

```
plot_time_series_boxplot(
  .data,
  .date_var,
  .value,
  .period,
  .color_var = NULL,
  .facet_vars = NULL,
  .facet_ncol = 1,
  .facet_nrow = 1,
  .facet_scales = "free_y",
  .facet_dir = "h",
  .facet_collapse = FALSE,
  .facet_collapse_sep = " ",
  .facet_strip_remove = FALSE,
  .line_color = "#2c3e50",
  .line_size = 0.5,
  .line_type = 1,
  .line_alpha = 1,
  .y_intercept = NULL,
  .y_intercept_color = "#2c3e50",
  .smooth = TRUE,
  .smooth_func = ~mean(.x, na.rm = TRUE),
  .smooth_period = "auto",
  .smooth_message = FALSE,
  .smooth_span = NULL,
  .smooth_degree = 2,
  .smooth_color = "#3366FF",
  .smooth_size = 1,
  .smooth_alpha = 1,
  .legend_show = TRUE,
```

```
  .title = "Time Series Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Legend",
  .interactive = TRUE,
  .plotly_slider = FALSE,
  .trelliscope = FALSE,
  .trelliscope_params = list()
)
```

Arguments

`.data`	A `tibble` or `data.frame` with a time-based column
`.date_var`	A column containing either date or date-time values
`.value`	A column containing numeric values
`.period`	A time series unit of aggregation for the boxplot. Examples include: • "1 week" • "3 years" • "30 minutes"
`.color_var`	A categorical column that can be used to change the line color
`.facet_vars`	One or more grouping columns that broken out into `ggplot2` facets. These can be selected using `tidyselect()` helpers (e.g `contains()`).
`.facet_ncol`	Number of facet columns.
`.facet_nrow`	Number of facet rows (only used for `.trelliscope = TRUE`)
`.facet_scales`	Control facet x & y-axis ranges. Options include "fixed", "free", "free_y", "free_x"
`.facet_dir`	The direction of faceting ("h" for horizontal, "v" for vertical). Default is "h".
`.facet_collapse`	Multiple facets included on one facet strip instead of multiple facet strips.
`.facet_collapse_sep`	The separator used for collapsing facets.
`.facet_strip_remove`	Whether or not to remove the strip and text label for each facet.
`.line_color`	Line color. Overrided if `.color_var` is specified.
`.line_size`	Line size.
`.line_type`	Line type.
`.line_alpha`	Line alpha (opacity). Range: (0, 1).
`.y_intercept`	Value for a y-intercept on the plot
`.y_intercept_color`	Color for the y-intercept
`.smooth`	Logical - Whether or not to include a trendline smoother. Uses See `smooth_vec()` to apply a LOESS smoother.

.smooth_func Defines how to aggregate the .value to show the smoothed trendline. The default is ~ mean(.x, na.rm = TRUE), which uses lambda function to ensure NA values are removed. Possible values are:
- A function, e.g. mean.
- A purrr-style lambda, e.g. ~ mean(.x, na.rm = TRUE)

.smooth_period Number of observations to include in the Loess Smoother. Set to "auto" by default, which uses tk_get_trend() to determine a logical trend cycle.

.smooth_message Logical. Whether or not to return the trend selected as a message. Useful for those that want to see what .smooth_period was selected.

.smooth_span Percentage of observations to include in the Loess Smoother. You can use either period or span. See smooth_vec().

.smooth_degree Flexibility of Loess Polynomial. Either 0, 1, 2 (0 = lest flexible, 2 = more flexible).

.smooth_color Smoother line color

.smooth_size Smoother line size

.smooth_alpha Smoother alpha (opacity). Range: (0, 1).

.legend_show Toggles on/off the Legend

.title Title for the plot

.x_lab X-axis label for the plot

.y_lab Y-axis label for the plot

.color_lab Legend label if a color_var is used.

.interactive Returns either a static (ggplot2) visualization or an interactive (plotly) visualization

.plotly_slider If TRUE, returns a plotly date range slider.

.trelliscope Returns either a normal plot or a trelliscopejs plot (great for many time series) Must have trelliscopejs installed.

.trelliscope_params Pass parameters to the trelliscopejs::facet_trelliscope() function as a list(). The only parameters that cannot be passed are:
- ncol: use .facet_ncol
- nrow: use .facet_nrow
- scales: use facet_scales
- as_plotly: use .interactive

Details

plot_time_series_boxplot() is a scalable function that works with both *ungrouped* and *grouped* data.frame objects (and tibbles!).

Interactive by Default

plot_time_series_boxplot() is built for exploration using:

- **Interactive Plots:** plotly (default) - Great for exploring!

- **Static Plots:** ggplot2 (set .interactive = FALSE) - Great for PDF Reports

By default, an interactive plotly visualization is returned.

Scalable with Facets & Dplyr Groups

plot_time_series_boxplot() returns multiple time series plots using ggplot2 facets:

- group_by() - If groups are detected, multiple facets are returned
- plot_time_series_boxplot(.facet_vars) - You can manually supply facets as well.

Can Transform Values just like ggplot

The .values argument accepts transformations just like ggplot2. For example, if you want to take the log of sales you can use a call like plot_time_series_boxplot(date, log(sales)) and the log transformation will be applied.

Smoother Period / Span Calculation

The .smooth = TRUE option returns a smoother that is calculated based on either:

1. A .smooth_func: The method of aggregation. Usually an aggregation like mean is used. The purrr-style function syntax can be used to apply complex functions.
2. A .smooth_period: Number of observations
3. A .smooth_span: A percentage of observations

By default, the .smooth_period is automatically calculated using 75% of the observertions. This is the same as geom_smooth(method = "loess", span = 0.75).

A user can specify a time-based window (e.g. .smooth_period = "1 year") or a numeric value (e.g. smooth_period = 365).

Time-based windows return the median number of observations in a window using tk_get_trend().

Value

A static ggplot2 plot or an interactive plotly plot

Examples

```
library(dplyr)
library(lubridate)

# Works with individual time series
FANG %>%
    filter(symbol == "FB") %>%
    plot_time_series_boxplot(
        date, adjusted,
        .period      = "3 month",
        .interactive = FALSE)

# Works with groups
FANG %>%
    group_by(symbol) %>%
    plot_time_series_boxplot(
        date, adjusted,
```

```
        .period      = "3 months",
        .facet_ncol  = 2,     # 2-column layout
        .interactive = FALSE)


# Can also group inside & use .color_var
FANG %>%
    mutate(year = year(date)) %>%
    plot_time_series_boxplot(
        date, adjusted,
        .period      = "3 months",
        .facet_vars   = c(symbol, year), # add groups/facets
        .color_var    = year,            # color by year
        .facet_ncol   = 4,
        .facet_scales = "free",
        .interactive  = FALSE)


# Can apply transformations to .value or .color_var
# - .value = log(adjusted)
# - .color_var = year(date)
FANG %>%
    plot_time_series_boxplot(
        date, log(adjusted),
        .period      = "3 months",
        .color_var    = year(date),
        .facet_vars   = contains("symbol"),
        .facet_ncol   = 2,
        .facet_scales = "free",
        .y_lab        = "Log Scale",
        .interactive  = FALSE)

# Can adjust the smoother
FANG %>%
    group_by(symbol) %>%
    plot_time_series_boxplot(
        date, adjusted,
        .period           = "3 months",
        .smooth           = TRUE,
        .smooth_func      = median,    # Smoother function
        .smooth_period    = "5 years", # Smoother Period
        .facet_ncol       = 2,
        .interactive      = FALSE)
```

---

plot_time_series_cv_plan

Visualize a Time Series Resample Plan

Description

The `plot_time_series_cv_plan()` function provides a visualization for a time series resample specification (`rset`) of either `rolling_origin` or `time_series_cv` class.

Usage

```
plot_time_series_cv_plan(
  .data,
  .date_var,
  .value,
  ...,
  .smooth = FALSE,
  .title = "Time Series Cross Validation Plan"
)
```

Arguments

`.data`	A time series resample specification of of either `rolling_origin` or `time_series_cv` class or a data frame (tibble) that has been prepared using `tk_time_series_cv_plan()`.
`.date_var`	A column containing either date or date-time values
`.value`	A column containing numeric values
`...`	Additional parameters passed to `plot_time_series()`
`.smooth`	Logical - Whether or not to include a trendline smoother. Uses See `smooth_vec()` to apply a LOESS smoother.
`.title`	Title for the plot

Details

Resample Set

A resample set is an output of the `timetk::time_series_cv()` function or the `rsample::rolling_origin()` function.

Value

Returns a static `ggplot` or interactive `plotly` object depending on whether or not `.interactive` is `FALSE` or `TRUE`, respectively.

See Also

- `time_series_cv()` and `rsample::rolling_origin()` - Functions used to create time series resample specfications.
- `plot_time_series_cv_plan()` - The plotting function used for visualizing the time series resample plan.

Examples

```
library(dplyr)
library(rsample)

FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

resample_spec <- time_series_cv(
    FB_tbl,
    initial = "1 year",
    assess  = "6 weeks",
    skip    = "3 months",
    lag     = "1 month",
    cumulative  = FALSE,
    slice_limit = 6
)

resample_spec %>% tk_time_series_cv_plan()

resample_spec %>%
    tk_time_series_cv_plan() %>%
    plot_time_series_cv_plan(
        date, adjusted, # date variable and value variable
        # Additional arguments passed to plot_time_series(),
        .facet_ncol = 2,
        .line_alpha = 0.5,
        .interactive = FALSE
    )
```

---

plot_time_series_regression *Visualize a Time Series Linear Regression Formula*

Description

A wrapper for `stats::lm()` that overlays a linear regression fitted model over a time series, which can help show the effect of feature engineering

Usage

```
plot_time_series_regression(
  .data,
  .date_var,
  .formula,
  .show_summary = FALSE,
  ...
)
```

Arguments

.data	A tibble or data.frame with a time-based column
.date_var	A column containing either date or date-time values
.formula	A linear regression formula. The left-hand side of the formula is used as the y-axis value. The right-hand side of the formula is used to develop the linear regression model. See stats::lm() for details.
.show_summary	If TRUE, prints the summary.lm().
...	Additional arguments passed to plot_time_series()

Details

plot_time_series_regression() is a scalable function that works with both *ungrouped* and *grouped* data.frame objects (and tibbles!).

Time Series Formula

The .formula uses stats::lm() to apply a linear regression, which is used to visualize the effect of feature engineering on a time series.

- The left-hand side of the formula is used as the y-axis value.
- The right-hand side of the formula is used to develop the linear regression model.

Interactive by Default

plot_time_series_regression() is built for exploration using:

- **Interactive Plots:** plotly (default) - Great for exploring!
- **Static Plots:** ggplot2 (set .interactive = FALSE) - Great for PDF Reports

By default, an interactive plotly visualization is returned.

Scalable with Facets & Dplyr Groups

plot_time_series_regression() returns multiple time series plots using ggplot2 facets:

- group_by() - If groups are detected, multiple facets are returned
- plot_time_series_regression(.facet_vars) - You can manually supply facets as well.

Value

A static ggplot2 plot or an interactive plotly plot

Examples

```
library(dplyr)
library(lubridate)

# ---- SINGLE SERIES ----
m4_monthly %>%
    filter(id == "M750") %>%
    plot_time_series_regression(
        .date_var     = date,
```

```
        .formula      = log(value) ~ as.numeric(date) + month(date, label = TRUE),
        .show_summary = TRUE,
        .facet_ncol   = 2,
        .interactive  = FALSE
    )


# ---- GROUPED SERIES ----
m4_monthly %>%
    group_by(id) %>%
    plot_time_series_regression(
        .date_var    = date,
        .formula     = log(value) ~ as.numeric(date) + month(date, label = TRUE),
        .facet_ncol  = 2,
        .interactive = FALSE
    )
```

---

set_tk_time_scale_template

Get and modify the Time Scale Template

Description

Get and modify the Time Scale Template

Usage

```
set_tk_time_scale_template(.data)

get_tk_time_scale_template()

tk_time_scale_template()
```

Arguments

`.data`	A `tibble` with a "time_scale", "frequency", and "trend" columns.

Details

Used to get and set the time scale template, which is used by `tk_get_frequency()` and `tk_get_trend()` when `period = "auto"`.

The predefined template is stored in a function `tk_time_scale_template()`. This is the default used by `timetk`.

Changing the Default Template

- You can access the current template with `get_tk_time_scale_template()`.
- You can modify the current template with `set_tk_time_scale_template()`.

Value

- `get_tk_time_scale_template()`: Returns `tibble` containing the time scale template information.
- `set_tk_time_scale_template()`: Returns nothing.

See Also

- Automated Frequency and Trend Calculation: `tk_get_frequency()`, `tk_get_trend()`

Examples

```
get_tk_time_scale_template()

set_tk_time_scale_template(tk_time_scale_template())
```

---

slice_period — *Apply slice inside periods (windows)*

Description

Applies a dplyr slice inside a time-based period (window).

Usage

```
slice_period(.data, ..., .date_var, .period = "1 day")
```

Arguments

`.data`	A `tbl` object or `data.frame`
`...`	For `slice()`: <`data-masking`> Integer row values. Provide either positive values to keep, or negative values to drop. The values provided must be either all positive or all negative. Indices beyond the number of rows in the input are silently ignored. For `slice_*()`, these arguments are passed on to methods.
`.date_var`	A column containing date or date-time values. If missing, attempts to auto-detect date column.
`.period`	A period to slice within. Time units are grouped using `lubridate::floor_date()` or `lubridate::ceiling_date()`. The value can be: • `second` • `minute` • `hour` • `day` • `week`

- month
- bimonth
- quarter
- season
- halfyear
- year

Arbitrary unique English abbreviations as in the `lubridate::period()` constructor are allowed:

- "1 year"
- "2 months"
- "30 seconds"

Value

A `tibble` or `data.frame`

See Also

Time-Based dplyr functions:

- `summarise_by_time()` - Easily summarise using a date column.
- `mutate_by_time()` - Simplifies applying mutations by time windows.
- `pad_by_time()` - Insert time series rows with regularly spaced timestamps
- `filter_by_time()` - Quickly filter using date ranges.
- `filter_period()` - Apply filtering expressions inside periods (windows)
- `slice_period()` - Apply slice inside periods (windows)
- `condense_period()` - Convert to a different periodicity
- `between_time()` - Range detection for date or date-time sequences.
- `slidify()` - Turn any function into a sliding (rolling) function

Examples

```
# Libraries
library(dplyr)

# First 5 observations in each month
m4_daily %>%
    group_by(id) %>%
    slice_period(1:5, .period = "1 month")

# Last observation in each month
m4_daily %>%
    group_by(id) %>%
    slice_period(n(), .period = "1 month")
```

`slidify` *Create a rolling (sliding) version of any function*

Description

`slidify` returns a rolling (sliding) version of the input function, with a rolling (sliding) `.period` specified by the user.

Usage

```
slidify(
  .f,
  .period = 1,
  .align = c("center", "left", "right"),
  .partial = FALSE,
  .unlist = TRUE
)
```

Arguments

`.f` A function, formula, or vector (not necessarily atomic).

If a **function**, it is used as is.

If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use `.`
- For a two argument function, use `.x` and `.y`
- For more arguments, use `..1`, `..2`, `..3` etc

This syntax allows you to create very compact anonymous functions. Note that formula functions conceptually take dots (that's why you can use `..1` etc). They silently ignore additional arguments that are not used in the formula expression.

If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.

`.period` The period size to roll over

`.align` One of "center", "left" or "right".

`.partial` Should the moving window be allowed to return partial (incomplete) windows instead of `NA` values. Set to `FALSE` by default, but can be switched to `TRUE` to remove `NA`'s.

`.unlist` If the function returns a single value each time it is called, use `.unlist = TRUE`. If the function returns more than one value, or a more complicated object (like a linear model), use `.unlist = FALSE` to create a list-column of the rolling results.

Details

The `slidify()` function is almost identical to `tibbletime::rollify()` with 3 improvements:

1. Alignment ("center", "left", "right")
2. Partial windows are allowed
3. Uses `slider` under the hood, which improves speed and reliability by implementing code at C++ level

Make any function a Sliding (Rolling) Function

`slidify()` turns a function into a sliding version of itself for use inside of a call to `dplyr::mutate()`, however it works equally as well when called from `purrr::map()`.

Because of it's intended use with `dplyr::mutate()`, `slidify` creates a function that always returns output with the same length of the input

Alignment

Rolling / Sliding functions generate `.period - 1` fewer values than the incoming vector. Thus, the vector needs to be aligned. Alignment of the vector follows 3 types:

- **center (default):** `NA` or `.partial` values are divided and added to the beginning and end of the series to "Center" the moving average. This is common in Time Series applications (e.g. denoising).
- **left:** `NA` or `.partial` values are added to the end to shift the series to the Left.
- **right:** `NA` or `.partial` values are added to the beginning to shift the series to the Right. This is common in Financial Applications (e.g moving average cross-overs).

Allowing Partial Windows

A key improvement over `tibbletime::slidify()` is that `timetk::slidify()` implements `.partial` rolling windows. Just set `.partial = TRUE`.

Value

A `function` with the rolling/sliding conversion applied.

References

- The Tibbletime R Package by Davis Vaughan, which includes the original `rollify()` Function

See Also

Transformation Functions:

- `slidify_vec()` - A simple vectorized function for applying summary functions to rolling windows.

Augmentation Functions (Add Rolling Multiple Columns):

- `tk_augment_slidify()` - For easily adding multiple rolling windows to you data

Slider R Package:

- `slider::pslide()` - The workhorse function that powers `timetk::slidify()`

Examples

```
library(dplyr)

FB <- FANG %>% dplyr::filter(symbol == "FB")


# --- ROLLING MEAN (SINGLE ARG EXAMPLE) ---

# Turn the normal mean function into a rolling mean with a 5 row .period
mean_roll_5 <- slidify(mean, .period = 5, .align = "right")

FB %>%
    mutate(rolling_mean_5 = mean_roll_5(adjusted))

# Use `partial = TRUE` to allow partial windows (those with less than the full .period)
mean_roll_5_partial <- slidify(mean, .period = 5, .align = "right", .partial = TRUE)

FB %>%
    mutate(rolling_mean_5 = mean_roll_5_partial(adjusted))

# There's nothing stopping you from combining multiple rolling functions with
# different .period sizes in the same mutate call

mean_roll_10 <- slidify(mean, .period = 10, .align = "right")

FB %>%
    select(symbol, date, adjusted) %>%
    mutate(
        rolling_mean_5  = mean_roll_5(adjusted),
        rolling_mean_10 = mean_roll_10(adjusted)
    )

# For summary operations like rolling means, we can accomplish large-scale
# multi-rolls with tk_augment_slidify()

FB %>%
    select(symbol, date, adjusted) %>%
    tk_augment_slidify(
        adjusted, .period = 5:10, .f = mean, .align = "right",
        .names = stringr::str_c("MA_", 5:10)
    )

# --- GROUPS AND ROLLING ----

# One of the most powerful things about this is that it works with
# groups since `mutate` is being used

mean_roll_3 <- slidify(mean, .period = 3, .align = "right")

FANG %>%
    group_by(symbol) %>%
    mutate(mean_roll = mean_roll_3(adjusted)) %>%
```

```
    slice(1:5)


# --- ROLLING CORRELATION (MULTIPLE ARG EXAMPLE) ---

# With 2 args, use the purrr syntax of ~ and .x, .y
# Rolling correlation example
cor_roll <- slidify(~cor(.x, .y), .period = 5, .align = "right")

FB %>%
    mutate(running_cor = cor_roll(adjusted, open))

# With >2 args, create an anonymous function with >2 args or use
# the purrr convention of ..1, ..2, ..3 to refer to the arguments
avg_of_avgs <- slidify(
    function(x, y, z) (mean(x) + mean(y) + mean(z)) / 3,
    .period = 10,
    .align = "right"
)

# Or
avg_of_avgs <- slidify(
    ~(mean(..1) + mean(..2) + mean(..3)) / 3,
    .period = 10,
    .align  = "right"
)

FB %>%
    mutate(avg_of_avgs = avg_of_avgs(open, high, low))

# Optional arguments MUST be passed at the creation of the rolling function
# Only data arguments that are "rolled over" are allowed when calling the
# rolling version of the function
FB$adjusted[1] <- NA

roll_mean_na_rm <- slidify(~mean(.x, na.rm = TRUE), .period = 5, .align = "right")

FB %>%
    mutate(roll_mean = roll_mean_na_rm(adjusted))


# --- ROLLING REGRESSIONS ----

# Rolling regressions are easy to implement using `.unlist = FALSE`
lm_roll <- slidify(~lm(.x ~ .y), .period = 90, .unlist = FALSE, .align = "right")

FB %>%
    tidyr::drop_na() %>%
    mutate(numeric_date = as.numeric(date)) %>%
    mutate(rolling_lm = lm_roll(adjusted, numeric_date)) %>%
    filter(!is.na(rolling_lm))
```

slidify_vec *Rolling Window Transformation*

Description

slidify_vec() applies a *summary function* to a rolling sequence of windows.

Usage

```
slidify_vec(
  .x,
  .f,
  ...,
  .period = 1,
  .align = c("center", "left", "right"),
  .partial = FALSE
)
```

Arguments

.x	A vector to have a rolling window transformation applied.
.f	A summary [function / formula] • If a **function**, e.g. mean, the function is used with any additional arguments, ... • If a **formula**, e.g. ~ mean(., na.rm = TRUE), it is converted to a function. This syntax allows you to create very compact anonymous functions.
...	Additional arguments passed on to the .f function.
.period	The number of periods to include in the local rolling window. This is effectively the "window size".
.align	One of "center", "left" or "right".
.partial	Should the moving window be allowed to return partial (incomplete) windows instead of NA values. Set to FALSE by default, but can be switched to TRUE to remove NA's.

Details

The slidify_vec() function is a wrapper for slider::slide_vec() with parameters simplified "center", "left", "right" alignment.

Vector Length In == Vector Length Out

NA values or .partial values are always returned to ensure the length of the return vector is the same length of the incoming vector. This ensures easier use with dplyr::mutate().

Alignment

Rolling functions generate `.period - 1` fewer values than the incoming vector. Thus, the vector needs to be aligned. Alignment of the vector follows 3 types:

- **Center:** `NA` or `.partial` values are divided and added to the beginning and end of the series to "Center" the moving average. This is common for de-noising operations. See also `[smooth_vec()]` for LOESS without NA values.
- **Left:** `NA` or `.partial` values are added to the end to shift the series to the Left.
- **Right:** `NA` or `.partial` values are added to the beginning to shif the series to the Right. This is common in Financial Applications such as moving average cross-overs.

Partial Values

- The advantage to using `.partial` values vs `NA` padding is that the series can be filled (good for time-series de-noising operations).
- The downside to partial values is that the partials can become less stable at the regions where incomplete windows are used.

If instability is not desirable for de-noising operations, a suitable alternative is `smooth_vec()`, which implements local polynomial regression.

Value

A numeric vector

References

- Slider R Package by Davis Vaughan

See Also

Modeling and More Complex Rolling Operations:

- `step_slidify()` - Roll apply for `tidymodels` modeling
- `tk_augment_slidify()` - Add many rolling columns group-wise
- `slidify()` - Turn any function into a rolling function. Great for rolling cor, rolling regression, etc.
- For more complex rolling operations, check out the `slider` R package.

Vectorized Transformation Functions:

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`

Examples

```
library(dplyr)
library(ggplot2)

# Training Data
FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

# ---- FUNCTION FORMAT ----
# - The `.f = mean` function is used. Argument `na.rm = TRUE` is passed as ...
FB_tbl %>%
    mutate(adjusted_30_ma = slidify_vec(
        .x      = adjusted,
        .period = 30,
        .f      = mean,
        na.rm   = TRUE,
        .align  = "center")) %>%
        ggplot(aes(date, adjusted)) +
        geom_line() +
        geom_line(aes(y = adjusted_30_ma), color = "blue", na.rm = TRUE)

# ---- FORMULA FORMAT ----
# - Anonymous function `.f = ~ mean(., na.rm = TRUE)` is used
FB_tbl %>%
    mutate(adjusted_30_ma = slidify_vec(
        .x      = adjusted,
        .period = 30,
        .f      = ~ mean(., na.rm = TRUE),
        .align  = "center")) %>%
        ggplot(aes(date, adjusted)) +
        geom_line() +
        geom_line(aes(y = adjusted_30_ma), color = "blue", na.rm = TRUE)

# ---- PARTIAL VALUES ----
# - set `.partial = TRUE`
FB_tbl %>%
    mutate(adjusted_30_ma = slidify_vec(
        .x       = adjusted,
        .f       = ~ mean(., na.rm = TRUE),
        .period  = 30,
        .align   = "center",
        .partial = TRUE)) %>%
        ggplot(aes(date, adjusted)) +
        geom_line() +
        geom_line(aes(y = adjusted_30_ma), color = "blue")

# ---- Loess vs Moving Average ----
# - Loess: Using `.degree = 0` to make less flexible. Comparable to a moving average.

FB_tbl %>%
    mutate(
```

```
        adjusted_loess_30 = smooth_vec(adjusted, period = 30, degree = 0),
        adjusted_ma_30    = slidify_vec(adjusted, .f = mean,
                                        .period = 30, .partial = TRUE)
    ) %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(aes(y = adjusted_loess_30), color = "red") +
    geom_line(aes(y = adjusted_ma_30), color = "blue") +
    labs(title = "Loess vs Moving Average")
```

`smooth_vec` *Smoothing Transformation using Loess*

Description

`smooth_vec()` applies a LOESS transformation to a numeric vector.

Usage

```
smooth_vec(x, period = 30, span = NULL, degree = 2)
```

Arguments

`x`	A numeric vector to have a smoothing transformation applied.
`period`	The number of periods to include in the local smoothing. Similar to window size for a moving average. See details for an explanation `period` vs `span` specification.
`span`	The span is a percentage of data to be included in the smoothing window. Period is preferred for shorter windows to fix the window size. See details for an explanation `period` vs `span` specification.
`degree`	The degree of the polynomials to be used. Accetable values (least to most flexible): 0, 1, 2. Set to 2 by default for 2nd order polynomial (most flexible).

Details

Benefits:

- When using `period`, the effect is **similar to a moving average without creating missing values.**
- When using `span`, the effect is to detect the trend in a series **using a percentage of the total number of observations.**

Loess Smoother Algorithm This function is a simplified wrapper for the `stats::loess()` with a modification to set a fixed `period` rather than a percentage of data points via a `span`.

Why Period vs Span? The `period` is fixed whereas the `span` changes as the number of observations change.

When to use Period? The effect of using a `period` is similar to a Moving Average where the Window Size is the **Fixed Period**. This helps when you are trying to smooth local trends. If you want a 30-day moving average, specify `period = 30`.

When to use Span? Span is easier to specify when you want a **Long-Term Trendline** where the window size is unknown. You can specify `span = 0.75` to locally regress using a window of 75% of the data.

Value

A numeric vector

See Also

Loess Modeling Functions:

- `step_smooth()` - Recipe for `tidymodels` workflow

Additional Vector Functions:

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`

Examples

```
library(dplyr)
library(ggplot2)

# Training Data
FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

# ---- PERIOD ----

FB_tbl %>%
    mutate(adjusted_30 = smooth_vec(adjusted, period = 30, degree = 2)) %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(aes(y = adjusted_30), color = "red")
```

```
# ---- SPAN ----

FB_tbl %>%
    mutate(adjusted_30 = smooth_vec(adjusted, span = 0.75, degree = 2)) %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(aes(y = adjusted_30), color = "red")

# ---- Loess vs Moving Average ----
# - Loess: Using `degree = 0` to make less flexible. Comperable to a moving average.

FB_tbl %>%
    mutate(
        adjusted_loess_30 = smooth_vec(adjusted, period = 30, degree = 0),
        adjusted_ma_30    = slidify_vec(adjusted, .period = 30,
                                        .f = mean, .partial = TRUE)
    ) %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(aes(y = adjusted_loess_30), color = "red") +
    geom_line(aes(y = adjusted_ma_30), color = "blue") +
    labs(title = "Loess vs Moving Average")
```

`standardize_vec` *Standardize to Mean 0, Standard Deviation 1 (Center & Scale)*

Description

Standardization is commonly used to center and scale numeric features to prevent one from dominating in algorithms that require data to be on the same scale.

Usage

```
standardize_vec(x, mean = NULL, sd = NULL, silent = FALSE)

standardize_inv_vec(x, mean, sd)
```

Arguments

`x`	A numeric vector.
`mean`	The mean used to invert the standardization
`sd`	The standard deviation used to invert the standardization process.
`silent`	Whether or not to report the automated `mean` and `sd` parameters as a message.

Details

Standardization vs Normalization

- **Standardization** refers to a transformation that reduces the range to mean 0, standard deviation 1
- **Normalization** refers to a transformation that reduces the min-max range: (0, 1)

Value

Returns a `numeric` vector with the standardization transformation applied.

See Also

- Normalization/Standardization: `standardize_vec()`, `normalize_vec()`
- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`, `ts_clean_vec()`

Examples

```
library(dplyr)

d10_daily <- m4_daily %>% dplyr::filter(id == "D10")

# --- VECTOR ----

value_std <- standardize_vec(d10_daily$value)
value     <- standardize_inv_vec(value_std,
                                 mean = 2261.60682492582,
                                 sd   = 175.603721730477)

# --- MUTATE ----

m4_daily %>%
    group_by(id) %>%
    mutate(value_std = standardize_vec(value))
```

step_box_cox *Box-Cox Transformation using Forecast Methods*

Description

step_box_cox creates a *specification* of a recipe step that will transform data using a Box-Cox transformation. This function differs from recipes::step_BoxCox by adding multiple methods including Guerrero lambda optimization and handling for negative data used in the Forecast R Package.

Usage

```
step_box_cox(
  recipe,
  ...,
  method = c("guerrero", "loglik"),
  limits = c(-1, 2),
  role = NA,
  trained = FALSE,
  lambdas_trained = NULL,
  skip = FALSE,
  id = rand_id("box_cox")
)

## S3 method for class 'step_box_cox'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more selector functions to choose which variables are affected by the step. See selections() for more details. For the tidy method, these are not currently used.
method	One of "guerrero" or "loglik"
limits	A length 2 numeric vector defining the range to compute the transformation parameter lambda.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
lambdas_trained	A numeric vector of transformation values. This is NULL until computed by prep().
skip	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome

variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.

`id` A character string that is unique to this step to identify it.

`x` A `step_box_cox` object.

Details

The `step_box_cox()` function is designed specifically to handle time series using methods implemented in the Forecast R Package.

Negative Data

This function can be applied to Negative Data.

Lambda Optimization Methods

This function uses 2 methods for optimizing the lambda selection from the Forecast R Package:

1. `method = "guerrero"`: Guerrero's (1993) method is used, where lambda minimizes the coefficient of variation for subseries of x.
2. `method = loglik`: the value of lambda is chosen to maximize the profile log likelihood of a linear model fitted to x. For non-seasonal data, a linear time trend is fitted while for seasonal data, a linear time trend with seasonal dummy variables is used.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected) and `value` (the lambda estimate).

References

1. Guerrero, V.M. (1993) Time-series analysis supported by power transformations. *Journal of Forecasting*, **12**, 37–48.
2. Box, G. E. P. and Cox, D. R. (1964) An analysis of transformations. *JRSS* B **26** 211–246.

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Transformations to reduce variance:

- `recipes::step_log()` - Log transformation

- `recipes::step_sqrt()` - Square-Root Power Transformation

Recipe Setup and Application:

- `recipes::recipe()`
- `recipes::prep()`
- `recipes::bake()`

Examples

```
library(dplyr)
library(recipes)

FANG_wide <- FANG %>%
    select(symbol, date, adjusted) %>%
    tidyr::pivot_wider(names_from = symbol, values_from = adjusted)

recipe_box_cox <- recipe(~ ., data = FANG_wide) %>%
    step_box_cox(FB, AMZN, NFLX, GOOG) %>%
    prep()

recipe_box_cox %>% bake(FANG_wide)

recipe_box_cox %>% tidy(1)
```

`step_diff` *Create a differenced predictor*

Description

`step_diff` creates a *specification* of a recipe step that will add new columns of differenced data. Differenced data will include NA values where a difference was induced. These can be removed with `step_naomit()`.

Usage

```
step_diff(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  lag = 1,
  difference = 1,
  log = FALSE,
  prefix = "diff_",
  columns = NULL,
  skip = FALSE,
```

```
  id = rand_id("diff")
)

## S3 method for class 'step_diff'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more selector functions to choose which variables are affected by the step. See `selections()` for more details.
role	Defaults to "predictor"
trained	A logical to indicate if the quantities for preprocessing have been estimated.
lag	A vector of positive integers identifying which lags (how far back) to be included in the differencing calculation.
difference	The number of differences to perform.
log	Calculates log differences instead of differences.
prefix	A prefix for generated column names, default to "diff_".
columns	A character string of variable names that will be populated (eventually) by the `terms` argument.
skip	A logical. Should the step be skipped when the recipe is baked by `bake.recipe()`? While all operations are baked when `prep.recipe()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations
id	A character string that is unique to this step to identify it.
x	A `step_diff` object.

Details

The step assumes that the data are already *in the proper sequential order* for lagging.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any).

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`

- Imputation: step_ts_impute(), step_ts_clean()
- Padding: step_ts_pad()

Remove NA Values:

- recipes::step_naomit()

Main Recipe Functions:

- recipes::recipe()
- recipes::prep()
- recipes::bake()

Examples

```
library(recipes)


FANG_wide <- FANG %>%
    dplyr::select(symbol, date, adjusted) %>%
    tidyr::pivot_wider(names_from = symbol, values_from = adjusted)


# Make and apply recipe ----

recipe_diff <- recipe(~ ., data = FANG_wide) %>%
  step_diff(FB, AMZN, NFLX, GOOG, lag = 1:3, difference = 1) %>%
  prep()

recipe_diff %>% bake(FANG_wide)


# Get information with tidy ----

recipe_diff %>% tidy()

recipe_diff %>% tidy(1)
```

---

step_fourier *Fourier Features for Modeling Seasonality*

Description

step_fourier creates a a *specification* of a recipe step that will convert a Date or Date-time column into a Fourier series

Usage

```
step_fourier(
  recipe,
  ...,
  period,
  K,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  scale_factor = NULL,
  skip = FALSE,
  id = rand_id("fourier")
)

## S3 method for class 'step_fourier'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	A single column with class `Date` or `POSIXct`. See `recipes::selections()` for more details. For the `tidy` method, these are not currently used.
period	The numeric period for the oscillation frequency. See details for examples of `period` specification.
K	The number of orders to include for each sine/cosine fourier series. More orders increase the number of fourier terms and therefore the variance of the fitted model at the expense of bias. See details for examples of `K` specification.
role	For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
columns	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `recipes::prep()` is used.
scale_factor	A factor for scaling the numeric index extracted from the date or date-time feature. This is a placeholder and will be populated once `recipes::prep()` is used.
skip	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.
id	A character string that is unique to this step to identify it.
x	A `step_fourier` object.

Details

Date Variable

Unlike other steps, `step_fourier` does *not* remove the original date variables. `recipes::step_rm()` can be used for this purpose.

Period Specification

The `period` argument is used to generate the distance between peaks in the fourier sequence. The key is to line up the peaks with unique seasonalities in the data.

For Daily Data, typical period specifications are:

- Yearly frequency is 365
- Quarterly frequency is 365 / 4 = 91.25
- Monthly frequency is 365 / 12 = 30.42

K Specification

The `K` argument specifies the maximum number of orders of Fourier terms. Examples:

- Specifying `period = 365` and `K = 1` will return a `cos365_K1` and `sin365_K1` fourier series
- Specifying `period = 365` and `K = 2` will return a `cos365_K1`, `cos365_K2`, `sin365_K1` and `sin365_K2` sequence, which tends to increase the models ability to fit vs the `K = 1` specification (at the expense of possibly overfitting).

Multiple values of `period` and `K`

It's possible to specify multiple values of `period` in a single step such as `step_fourier(period = c(91.25, 365), K = 2`. This returns 8 Fouriers series:

- `cos91.25_K1`, `sin91.25_K1`, `cos91.25_K2`, `sin91.25_K2`
- `cos365_K1`, `sin365_K1`, `cos365_K2`, `sin365_K2`

Value

For `step_fourier`, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected), `value` (the feature names).

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Main Recipe Functions:

- `recipes::recipe()`
- `recipes::prep()`
- `recipes::bake()`

Examples

```
library(recipes)
library(dplyr)

FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

# Create a recipe object with a timeseries signature step
# - 252 Trade days per year
# - period = c(252/4, 252): Adds quarterly and yearly fourier series
# - K = 2: Adds 1st and 2nd fourier orders

rec_obj <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_fourier(date, period = c(252/4, 252), K = 2)

# View the recipe object
rec_obj

# Prepare the recipe object
prep(rec_obj)

# Bake the recipe object - Adds the Fourier Series
bake(prep(rec_obj), FB_tbl)

# Tidy shows which features have been added during the 1st step
#  in this case, step 1 is the step_timeseries_signature step
tidy(prep(rec_obj))
tidy(prep(rec_obj), number = 1)
```

`step_holiday_signature` *Holiday Feature (Signature) Generator*

Description

`step_holiday_signature` creates a a *specification* of a recipe step that will convert date or date-time data into many holiday features that can aid in machine learning with time-series data. By default, many features are returned for different *holidays, locales, and stock exchanges*.

Usage

```
step_holiday_signature(
  recipe,
  ...,
  holiday_pattern = ".",
  locale_set = "all",
  exchange_set = "all",
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  features = NULL,
  skip = FALSE,
  id = rand_id("holiday_signature")
)

## S3 method for class 'step_holiday_signature'
tidy(x, ...)
```

Arguments

`recipe`	A recipe object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose which variables that will be used to create the new variables. The selected variables should have class `Date` or `POSIXct`. See `recipes::selections()` for more details. For the `tidy` method, these are not currently used.
`holiday_pattern`	A regular expression pattern to search the "Holiday Set".
`locale_set`	Return binary holidays based on locale. One of: "all", "none", "World", "US", "CA", "GB", "FR", "IT", "JP", "CH", "DE".
`exchange_set`	Return binary holidays based on Stock Exchange Calendars. One of: "all", "none", "NYSE", "LONDON", "NERC", "TSX", "ZURICH".
`role`	For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `recipes::prep()` is used.
`features`	A character string of features that will be generated. This field is a placeholder and will be populated once `recipes::prep()` is used.
`skip`	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.

id	A character string that is unique to this step to identify it.
x	A step_holiday_signature object.

Details

Use Holiday Pattern and Feature Sets to Pare Down Features By default, you're going to get A LOT of Features. This is a good thing because many machine learning algorithms have regularization built in. But, in many cases you will still want to reduce the number of *unnecessary features*. Here's how:

- **Holiday Pattern:** This is a Regular Expression pattern that can be used to filter. Try holiday_pattern = "(US_Christ)|(US_Thanks)" to return just Christmas and Thanksgiving features.
- **Locale Sets:** This is a logical as to whether or not the locale has a holiday. For locales outside of US you may want to combine multiple locales. For example, locale_set = c("World", "GB") returns both World Holidays and Great Britain.
- **Exchange Sets:** This is a logical as to whether or not the *Business is off* due to a holiday. Different Stock Exchanges are used as a proxy for business holiday calendars. For example, exchange_set = "NYSE" returns business holidays for New York Stock Exchange.

Removing Unnecessary Features By default, many features are created automatically. Unnecessary features can be removed using recipes::step_rm() and recipes::selections() for more details.

Value

For step_holiday_signature, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns terms (the selectors or variables selected), value (the feature names).

See Also

Time Series Analysis:

- Engineered Features: step_timeseries_signature(), step_holiday_signature(), step_fourier()
- Diffs & Lags step_diff(), recipes::step_lag()
- Smoothing: step_slidify(), step_smooth()
- Variance Reduction: step_box_cox()
- Imputation: step_ts_impute(), step_ts_clean()
- Padding: step_ts_pad()

Main Recipe Functions:

- recipes::recipe()
- recipes::prep()
- recipes::bake()

Examples

```
library(recipes)
library(dplyr)

# Sample Data
dates_in_2017_tbl <- tibble::tibble(
    index = tk_make_timeseries("2017-01-01", "2017-12-31", by = "day")
)

# Add US holidays and Non-Working Days due to Holidays
# - Physical Holidays are added with holiday pattern (individual) and locale_set
rec_holiday <- recipe(~ ., dates_in_2017_tbl) %>%
    step_holiday_signature(index,
                           holiday_pattern = "^US_",
                           locale_set      = "US",
                           exchange_set    = "NYSE")

# Not yet prep'ed - just returns parameters selected
rec_holiday %>% tidy(1)

# Prep the recipe
rec_holiday_prep <- prep(rec_holiday)

# Now prep'ed - returns new features that will be created
rec_holiday_prep %>% tidy(1)

# Apply the recipe to add new holiday features!
bake(rec_holiday_prep, dates_in_2017_tbl)
```

---

step_log_interval *Log Interval Transformation for Constrained Interval Forecasting*

Description

step_log_interval creates a *specification* of a recipe step that will transform data using a Log-Inerval transformation. This function provides a recipes interface for the log_interval_vec() transformation function.

Usage

```
step_log_interval(
  recipe,
  ...,
  limit_lower = "auto",
  limit_upper = "auto",
```

```
  offset = 0,
  role = NA,
  trained = FALSE,
  limit_lower_trained = NULL,
  limit_upper_trained = NULL,
  skip = FALSE,
  id = rand_id("log_interval")
)

## S3 method for class 'step_log_interval'
tidy(x, ...)
```

Arguments

- `recipe` — A `recipe` object. The step will be added to the sequence of operations for this recipe.
- `...` — One or more selector functions to choose which variables are affected by the step. See `selections()` for more details. For the `tidy` method, these are not currently used.
- `limit_lower` — A lower limit. Must be less than the minimum value. If set to "auto", selects zero.
- `limit_upper` — An upper limit. Must be greater than the maximum value. If set to "auto", selects a value that is 10% greater than the maximum value.
- `offset` — An offset to include in the log transformation. Useful when the data contains values less than or equal to zero.
- `role` — Not used by this step since no new variables are created.
- `trained` — A logical to indicate if the quantities for preprocessing have been estimated.
- `limit_lower_trained` — A numeric vector of transformation values. This is `NULL` until computed by `prep()`.
- `limit_upper_trained` — A numeric vector of transformation values. This is `NULL` until computed by `prep()`.
- `skip` — A logical. Should the step be skipped when the recipe is baked by `bake.recipe()`? While all operations are baked when `prep.recipe()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
- `id` — A character string that is unique to this step to identify it.
- `x` — A `step_log_interval` object.

Details

The `step_log_interval()` function is designed specifically to handle time series using methods implemented in the Forecast R Package.

Positive Data

If data includes values of zero, use `offset` to adjust the series to make the values positive.

Implementation

Refer to the `log_interval_vec()` function for the transformation implementation details.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected) and `value` (the lambda estimate).

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_log_interval()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Transformations to reduce variance:

- `recipes::step_log()` - Log transformation
- `recipes::step_sqrt()` - Square-Root Power Transformation

Recipe Setup and Application:

- `recipes::recipe()`
- `recipes::prep()`
- `recipes::bake()`

Examples

```
library(dplyr)
library(recipes)

FANG_wide <- FANG %>%
    select(symbol, date, adjusted) %>%
    tidyr::pivot_wider(names_from = symbol, values_from = adjusted)

recipe_log_interval <- recipe(~ ., data = FANG_wide) %>%
    step_log_interval(FB, AMZN, NFLX, GOOG, offset = 1) %>%
    prep()

recipe_log_interval %>%
    bake(FANG_wide) %>%
    tidyr::pivot_longer(-date) %>%
    plot_time_series(date, value, name, .smooth = FALSE, .interactive = FALSE)
```

```
recipe_log_interval %>% tidy(1)
```

---

step_slidify *Slidify Rolling Window Transformation*

Description

step_slidify creates a a *specification* of a recipe step that will apply a function to one or more a Numeric column(s).

Usage

```
step_slidify(
  recipe,
  ...,
  period,
  .f,
  align = c("center", "left", "right"),
  partial = FALSE,
  names = NULL,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  f_name = NULL,
  skip = FALSE,
  id = rand_id("slidify")
)

## S3 method for class 'step_slidify'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	One or more numeric columns to be smoothed. See recipes::selections() for more details. For the tidy method, these are not currently used.
period	The number of periods to include in the local rolling window. This is effectively the "window size".
.f	A summary **formula** in one of the following formats: • mean with no arguments • function(x) mean(x, na.rm = TRUE) • ~ mean(.x, na.rm = TRUE), it is converted to a function.

align
: Rolling functions generate `period - 1` fewer values than the incoming vector. Thus, the vector needs to be aligned. Alignment of the vector follows 3 types:

  - **Center:** `NA` or `.partial` values are divided and added to the beginning and end of the series to "Center" the moving average. This is common for de-noising operations. See also `[smooth_vec()]` for LOESS without NA values.
  - **Left:** `NA` or `.partial` values are added to the end to shift the series to the Left.
  - **Right:** `NA` or `.partial` values are added to the beginning to shif the series to the Right. This is common in Financial Applications such as moving average cross-overs.

partial
: Should the moving window be allowed to return partial (incomplete) windows instead of NA values. Set to FALSE by default, but can be switched to TRUE to remove NA's.

names
: An optional character string that is the same length of the number of terms selected by `terms`. These will be the names of the **new columns** created by the step.

  - If `NULL`, existing columns are transformed.
  - If not `NULL`, new columns will be created.

role
: For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.

trained
: A logical to indicate if the quantities for preprocessing have been estimated.

columns
: A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `recipes::prep()` is used.

f_name
: A character string for the function being applied. This field is a placeholder and will be populated during the `tidy()` step.

skip
: A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.

id
: A character string that is unique to this step to identify it.

x
: A `step_slidify` object.

Details

Alignment

Rolling functions generate `period - 1` fewer values than the incoming vector. Thus, the vector needs to be aligned. Alignment of the vector follows 3 types:

- **Center:** `NA` or `partial` values are divided and added to the beginning and end of the series to "Center" the moving average. This is common for de-noising operations. See also `[smooth_vec()]` for LOESS without NA values.
- **Left:** `NA` or `partial` values are added to the end to shift the series to the Left.

- **Right:** NA or partial values are added to the beginning to shif the series to the Right. This is common in Financial Applications such as moving average cross-overs.

Partial Values

- The advantage to using partial values vs NA padding is that the series can be filled (good for time-series de-noising operations).
- The downside to partial values is that the partials can become less stable at the regions where incomplete windows are used.

If instability is not desirable for de-noising operations, a suitable alternative is step_smooth(), which implements local polynomial regression.

Value

For step_slidify, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns terms (the selectors or variables selected), value (the feature names).

See Also

Time Series Analysis:

- Engineered Features: step_timeseries_signature(), step_holiday_signature(), step_fourier()
- Diffs & Lags step_diff(), recipes::step_lag()
- Smoothing: step_slidify(), step_smooth()
- Variance Reduction: step_box_cox()
- Imputation: step_ts_impute(), step_ts_clean()
- Padding: step_ts_pad()

Main Recipe Functions:

- recipes::recipe()
- recipes::prep()
- recipes::bake()

Examples

```
library(recipes)
library(dplyr)
library(ggplot2)

# Training Data
FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

# New Data - Make some fake new data next 90 time stamps
new_data <- FB_tbl %>%
    tail(90) %>%
```

```
    mutate(date = date %>% tk_make_future_timeseries(length_out = 90))

# OVERWRITE EXISTING COLUMNS -----

# Create a recipe object with a step_slidify
rec_ma_50 <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_slidify(adjusted, period = 50, .f = ~ mean(.x))

# Bake the recipe object - Applies the Moving Average Transformation
training_data_baked <- bake(prep(rec_ma_50), FB_tbl)

# Apply to New Data
new_data_baked <- bake(prep(rec_ma_50), new_data)

# Visualize effect
training_data_baked %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(color = "red", data = new_data_baked)

# ---- NEW COLUMNS ----
# Use the `names` argument to create new columns instead of overwriting existing

rec_ma_30_names <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_slidify(adjusted, period = 30, .f = mean, names = "adjusted_ma_30")

bake(prep(rec_ma_30_names), FB_tbl) %>%
    ggplot(aes(date, adjusted)) +
    geom_line(alpha = 0.5) +
    geom_line(aes(y = adjusted_ma_30), color = "red", size = 1)
```

---

step_slidify_augment   *Slidify Rolling Window Transformation (Augmented Version)*

Description

`step_slidify_augment` creates a a *specification* of a recipe step that will "augment" (add multiple new columns) that have had a sliding function applied.

Usage

```
step_slidify_augment(
  recipe,
  ...,
  period,
  .f,
  align = c("center", "left", "right"),
```

```
  partial = FALSE,
  prefix = "slidify_",
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  f_name = NULL,
  skip = FALSE,
  id = rand_id("slidify_augment")
)

## S3 method for class 'step_slidify_augment'
tidy(x, ...)
```

Arguments

`recipe` A recipe object. The step will be added to the sequence of operations for this recipe.

`...` One or more numeric columns to be smoothed. See `recipes::selections()` for more details. For the `tidy` method, these are not currently used.

`period` The number of periods to include in the local rolling window. This is effectively the "window size".

`.f` A summary **formula** in one of the following formats:

- `mean` with no arguments
- `function(x) mean(x, na.rm = TRUE)`
- `~ mean(.x, na.rm = TRUE)`, it is converted to a function.

`align` Rolling functions generate `period - 1` fewer values than the incoming vector. Thus, the vector needs to be aligned. Alignment of the vector follows 3 types:

- **Center:** `NA` or `.partial` values are divided and added to the beginning and end of the series to "Center" the moving average. This is common for de-noising operations. See also `[smooth_vec()]` for LOESS without NA values.
- **Left:** `NA` or `.partial` values are added to the end to shift the series to the Left.
- **Right:** `NA` or `.partial` values are added to the beginning to shif the series to the Right. This is common in Financial Applications such as moving average cross-overs.

`partial` Should the moving window be allowed to return partial (incomplete) windows instead of NA values. Set to FALSE by default, but can be switched to TRUE to remove NA's.

`prefix` A prefix for generated column names, default to "slidify_".

`role` For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.

`trained` A logical to indicate if the quantities for preprocessing have been estimated.

columns	A character string of variable names that will be populated (eventually) by the terms argument.
f_name	A character string for the function being applied. This field is a placeholder and will be populated during the tidy() step.
skip	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations
id	A character string that is unique to this step to identify it.
x	A step_slidify_augment object.

Details

Alignment

Rolling functions generate period - 1 fewer values than the incoming vector. Thus, the vector needs to be aligned. Alignment of the vector follows 3 types:

- **Center:** NA or partial values are divided and added to the beginning and end of the series to "Center" the moving average. This is common for de-noising operations. See also [smooth_vec()] for LOESS without NA values.
- **Left:** NA or partial values are added to the end to shift the series to the Left.
- **Right:** NA or partial values are added to the beginning to shif the series to the Right. This is common in Financial Applications such as moving average cross-overs.

Partial Values

- The advantage to using partial values vs NA padding is that the series can be filled (good for time-series de-noising operations).
- The downside to partial values is that the partials can become less stable at the regions where incomplete windows are used.

If instability is not desirable for de-noising operations, a suitable alternative is step_smooth(), which implements local polynomial regression.

Value

For step_slidify_augment, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns terms (the selectors or variables selected), value (the feature names).

See Also

Time Series Analysis:

- Engineered Features: step_timeseries_signature(), step_holiday_signature(), step_fourier()
- Diffs & Lags step_diff(), recipes::step_lag()
- Smoothing: step_slidify(), step_smooth()

- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Main Recipe Functions:

- `recipes::recipe()`
- `recipes::prep()`
- `recipes::bake()`

Examples

```
# library(tidymodels)
library(dplyr)
library(recipes)
library(parsnip)

m750 <- m4_monthly %>%
    filter(id == "M750") %>%
    mutate(value_2 = value / 2)

m750_splits <- time_series_split(m750, assess = "2 years", cumulative = TRUE)

# Make a recipe
recipe_spec <- recipe(value ~ date + value_2, rsample::training(m750_splits)) %>%
    step_slidify_augment(
        value, value_2,
        period = c(6, 12, 24),
        .f = ~ mean(.x),
        align = "center",
        partial = FALSE
    )

recipe_spec %>% prep() %>% juice()

bake(prep(recipe_spec), rsample::testing(m750_splits))
```

`step_smooth` *Smoothing Transformation using Loess*

Description

`step_smooth` creates a a *specification* of a recipe step that will apply local polynomial regression to one or more a Numeric column(s). The effect is smoothing the time series **similar to a moving average without creating missing values or using partial smoothing.**

Usage

```
step_smooth(
  recipe,
  ...,
  period = 30,
  span = NULL,
  degree = 2,
  names = NULL,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("smooth")
)

## S3 method for class 'step_smooth'
tidy(x, ...)
```

Arguments

`recipe` A recipe object. The step will be added to the sequence of operations for this recipe.

`...` One or more numeric columns to be smoothed. See `recipes::selections()` for more details. For the `tidy` method, these are not currently used.

`period` The number of periods to include in the local smoothing. Similar to window size for a moving average. See details for an explanation `period` vs `span` specification.

`span` The span is a percentage of data to be included in the smoothing window. Period is preferred for shorter windows to fix the window size. See details for an explanation `period` vs `span` specification.

`degree` The degree of the polynomials to be used. Set to 2 by default for 2nd order polynomial.

`names` An optional character string that is the same length of the number of terms selected by `terms`. These will be the names of the **new columns** created by the step.

- If `NULL`, existing columns are transformed.
- If not `NULL`, new columns will be created.

`role` For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.

`trained` A logical to indicate if the quantities for preprocessing have been estimated.

`columns` A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `recipes::prep()` is used.

`skip` A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.

`id` A character string that is unique to this step to identify it.

`x` A `step_smooth` object.

Details

Smoother Algorithm This function is a `recipe` specification that wraps the `stats::loess()` with a modification to set a fixed `period` rather than a percentage of data points via a `span`.

Why Period vs Span? The `period` is fixed whereas the `span` changes as the number of observations change.

When to use Period? The effect of using a `period` is similar to a Moving Average where the Window Size is the **Fixed Period**. This helps when you are trying to smooth local trends. If you want a 30-day moving average, specify `period = 30`.

When to use Span? Span is easier to specify when you want a **Long-Term Trendline** where the window size is unknown. You can specify `span = 0.75` to locally regress using a window of 75% of the data.

Warning - Using Span with New Data When using span on New Data, the number of observations is likely different than what you trained with. This means the trendline / smoother can be vastly different than the smoother you trained with.

Solution to Span with New Data Don't use `span`. Rather, use `period` to fix the window size. This ensures that new data includes the same number of observations in the local polynomial regression (loess) as the training data.

Value

For `step_smooth`, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected), `value` (the feature names).

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Main Recipe Functions:

- `recipes::recipe()`
- `recipes::prep()`
- `recipes::bake()`

Examples

```
library(recipes)
library(dplyr)
library(ggplot2)

# Training Data
FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

# New Data - Make some fake new data next 90 time stamps
new_data <- FB_tbl %>%
    tail(90) %>%
    mutate(date = date %>% tk_make_future_timeseries(length_out = 90))

# ---- PERIOD ----

# Create a recipe object with a step_smooth()
rec_smooth_period <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_smooth(adjusted, period = 30)

# Bake the recipe object - Applies the Loess Transformation
training_data_baked <- bake(prep(rec_smooth_period), FB_tbl)

# "Period" Effect on New Data
new_data_baked <- bake(prep(rec_smooth_period), new_data)

# Smoother's fit on new data is very similar because
# 30 days are used in the new data regardless of the new data being 90 days
training_data_baked %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(color = "red", data = new_data_baked)

# ---- SPAN ----

# Create a recipe object with a step_smooth
rec_smooth_span <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_smooth(adjusted, span = 0.03)

# Bake the recipe object - Applies the Loess Transformation
training_data_baked <- bake(prep(rec_smooth_span), FB_tbl)

# "Period" Effect on New Data
new_data_baked <- bake(prep(rec_smooth_span), new_data)

# Smoother's fit is not the same using span because new data is only 90 days
# and 0.03 x 90 = 2.7 days
training_data_baked %>%
    ggplot(aes(date, adjusted)) +
    geom_line() +
    geom_line(color = "red", data = new_data_baked)
```

```
# ---- NEW COLUMNS ----
# Use the `names` argument to create new columns instead of overwriting existing

rec_smooth_names <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_smooth(adjusted, period = 30, names = "adjusted_smooth_30") %>%
    step_smooth(adjusted, period = 180, names = "adjusted_smooth_180") %>%
    step_smooth(adjusted, span = 0.75, names = "long_term_trend")

bake(prep(rec_smooth_names), FB_tbl) %>%
    ggplot(aes(date, adjusted)) +
    geom_line(alpha = 0.5) +
    geom_line(aes(y = adjusted_smooth_30), color = "red", size = 1) +
    geom_line(aes(y = adjusted_smooth_180), color = "blue", size = 1) +
    geom_line(aes(y = long_term_trend), color = "orange", size = 1)
```

---

step_timeseries_signature *Time Series Feature (Signature) Generator*

Description

step_timeseries_signature creates a a *specification* of a recipe step that will convert date or date-time data into many features that can aid in machine learning with time-series data

Usage

```
step_timeseries_signature(
  recipe,
  ...,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("timeseries_signature")
)

## S3 method for class 'step_timeseries_signature'
tidy(x, ...)
```

Arguments

recipe A recipe object. The step will be added to the sequence of operations for this recipe.

...	One or more selector functions to choose which variables that will be used to create the new variables. The selected variables should have class `Date` or `POSIXct`. See `recipes::selections()` for more details. For the `tidy` method, these are not currently used.
`role`	For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`columns`	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once `recipes::prep()` is used.
`skip`	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.
`x`	A `step_timeseries_signature` object.

Details

Date Variable Unlike other steps, `step_timeseries_signature` does *not* remove the original date variables. `recipes::step_rm()` can be used for this purpose.

Scaling index.num The `index.num` feature created has a large magnitude (number of seconds since 1970-01-01). It's a good idea to scale and center this feature (e.g. use `recipes::step_normalize()`).

Removing Unnecessary Features By default, many features are created automatically. Unnecessary features can be removed using `recipes::step_rm()`.

Value

For `step_timeseries_signature`, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected), `value` (the feature names).

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Main Recipe Functions:

- recipes::recipe()
- recipes::prep()
- recipes::bake()

Examples

```
library(recipes)
library(dplyr)

FB_tbl <- FANG %>% dplyr::filter(symbol == "FB")

# Create a recipe object with a timeseries signature step
rec_obj <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_timeseries_signature(date)

# View the recipe object
rec_obj

# Prepare the recipe object
prep(rec_obj)

# Bake the recipe object - Adds the Time Series Signature
bake(prep(rec_obj), FB_tbl)

# Tidy shows which features have been added during the 1st step
#  in this case, step 1 is the step_timeseries_signature step
tidy(rec_obj)
tidy(rec_obj, number = 1)
```

---

step_ts_clean *Clean Outliers and Missing Data for Time Series*

Description

step_ts_clean creates a *specification* of a recipe step that will clean outliers and impute time series data.

Usage

```
step_ts_clean(
  recipe,
  ...,
  period = 1,
  lambda = "auto",
  role = NA,
  trained = FALSE,
```

```
  lambdas_trained = NULL,
  skip = FALSE,
  id = rand_id("ts_clean")
)

## S3 method for class 'step_ts_clean'
tidy(x, ...)
```

Arguments

`recipe`	A `recipe` object. The step will be added to the sequence of operations for this recipe.
`...`	One or more selector functions to choose which variables are affected by the step. See `selections()` for more details. For the `tidy` method, these are not currently used.
`period`	A seasonal period to use during the transformation. If `period = 1`, linear interpolation is performed. If `period > 1`, a robust STL decomposition is first performed and a linear interpolation is applied to the seasonally adjusted data.
`lambda`	A box cox transformation parameter. If set to `"auto"`, performs automated lambda selection.
`role`	Not used by this step since no new variables are created.
`trained`	A logical to indicate if the quantities for preprocessing have been estimated.
`lambdas_trained`	A named numeric vector of lambdas. This is `NULL` until computed by `recipes::prep()`. Note that, if the original data are integers, the mean will be converted to an integer to maintain the same a data type.
`skip`	A logical. Should the step be skipped when the recipe is baked by `bake.recipe()`? While all operations are baked when `prep.recipe()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
`id`	A character string that is unique to this step to identify it.
`x`	A `step_ts_clean` object.

Details

The `step_ts_clean()` function is designed specifically to handle time series using seasonal outlier detection methods implemented in the Forecast R Package.

Cleaning Outliers

#' Outliers are replaced with missing values using the following methods:

1. Non-Seasonal (`period = 1`): Uses `stats::supsmu()`
2. Seasonal (`period > 1`): Uses `forecast::mstl()` with `robust = TRUE` (robust STL decomposition) for seasonal series.

Imputation using Linear Interpolation

Three circumstances cause strictly linear interpolation:

1. **Period is 1:** With `period = 1`, a seasonality cannot be interpreted and therefore linear is used.
2. **Number of Non-Missing Values is less than 2-Periods**: Insufficient values exist to detect seasonality.
3. **Number of Total Values is less than 3-Periods**: Insufficient values exist to detect seasonality.

Seasonal Imputation using Linear Interpolation

For seasonal series with `period > 1`, a robust Seasonal Trend Loess (STL) decomposition is first computed. Then a linear interpolation is applied to the seasonally adjusted data, and the seasonal component is added back.

Box Cox Transformation

In many circumstances, a Box Cox transformation can help. Especially if the series is multiplicative meaning the variance grows exponentially. A Box Cox transformation can be automated by setting `lambda = "auto"` or can be specified by setting `lambda = numeric value`.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected) and `value` (the lambda estimate).

References

- Forecast R Package
- Forecasting Principles & Practices: Dealing with missing values and outliers

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Examples

```
library(dplyr)
library(tidyr)
library(recipes)

# Get missing values
FANG_wide <- FANG %>%
```

```
    select(symbol, date, adjusted) %>%
    pivot_wider(names_from = symbol, values_from = adjusted) %>%
    pad_by_time()

FANG_wide

# Apply Imputation
recipe_box_cox <- recipe(~ ., data = FANG_wide) %>%
    step_ts_clean(FB, AMZN, NFLX, GOOG, period = 252) %>%
    prep()

recipe_box_cox %>% bake(FANG_wide)

# Lambda parameter used during imputation process
recipe_box_cox %>% tidy(1)
```

---

step_ts_impute *Missing Data Imputation for Time Series*

Description

step_ts_impute creates a *specification* of a recipe step that will impute time series data.

Usage

```
step_ts_impute(
  recipe,
  ...,
  period = 1,
  lambda = NULL,
  role = NA,
  trained = FALSE,
  lambdas_trained = NULL,
  skip = FALSE,
  id = rand_id("ts_impute")
)

## S3 method for class 'step_ts_impute'
tidy(x, ...)
```

Arguments

recipe A recipe object. The step will be added to the sequence of operations for this recipe.

...	One or more selector functions to choose which variables are affected by the step. See `selections()` for more details. For the `tidy` method, these are not currently used.
period	A seasonal period to use during the transformation. If `period = 1`, linear interpolation is performed. If `period > 1`, a robust STL decomposition is first performed and a linear interpolation is applied to the seasonally adjusted data.
lambda	A box cox transformation parameter. If set to `"auto"`, performs automated lambda selection.
role	Not used by this step since no new variables are created.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
lambdas_trained	A named numeric vector of lambdas. This is `NULL` until computed by `recipes::prep()`. Note that, if the original data are integers, the mean will be converted to an integer to maintain the same a data type.
skip	A logical. Should the step be skipped when the recipe is baked by `bake.recipe()`? While all operations are baked when `prep.recipe()` is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using `skip = TRUE` as it may affect the computations for subsequent operations.
id	A character string that is unique to this step to identify it.
x	A `step_ts_impute` object.

Details

The `step_ts_impute()` function is designed specifically to handle time series

Imputation using Linear Interpolation

Three circumstances cause strictly linear interpolation:

1. **Period is 1:** With `period = 1`, a seasonality cannot be interpreted and therefore linear is used.
2. **Number of Non-Missing Values is less than 2-Periods**: Insufficient values exist to detect seasonality.
3. **Number of Total Values is less than 3-Periods**: Insufficient values exist to detect seasonality.

Seasonal Imputation using Linear Interpolation

For seasonal series with `period > 1`, a robust Seasonal Trend Loess (STL) decomposition is first computed. Then a linear interpolation is applied to the seasonally adjusted data, and the seasonal component is added back.

Box Cox Transformation

In many circumstances, a Box Cox transformation can help. Especially if the series is multiplicative meaning the variance grows exponentially. A Box Cox transformation can be automated by setting `lambda = "auto"` or can be specified by setting `lambda = numeric value`.

Value

An updated version of `recipe` with the new step added to the sequence of existing steps (if any). For the `tidy` method, a tibble with columns `terms` (the selectors or variables selected) and `value` (the lambda estimate).

References

- Forecast R Package
- Forecasting Principles & Practices: Dealing with missing values and outliers

See Also

Time Series Analysis:

- Engineered Features: `step_timeseries_signature()`, `step_holiday_signature()`, `step_fourier()`
- Diffs & Lags `step_diff()`, `recipes::step_lag()`
- Smoothing: `step_slidify()`, `step_smooth()`
- Variance Reduction: `step_box_cox()`
- Imputation: `step_ts_impute()`, `step_ts_clean()`
- Padding: `step_ts_pad()`

Recipe Setup and Application:

- `recipes::recipe()`
- `recipes::prep()`
- `recipes::bake()`

Examples

```
library(dplyr)
library(recipes)

# Get missing values
FANG_wide <- FANG %>%
    select(symbol, date, adjusted) %>%
    tidyr::pivot_wider(names_from = symbol, values_from = adjusted) %>%
    pad_by_time()

FANG_wide

# Apply Imputation
recipe_box_cox <- recipe(~ ., data = FANG_wide) %>%
    step_ts_impute(FB, AMZN, NFLX, GOOG, period = 252, lambda = "auto") %>%
    prep()

recipe_box_cox %>% bake(FANG_wide)

# Lambda parameter used during imputation process
recipe_box_cox %>% tidy(1)
```

step_ts_pad *Pad: Add rows to fill gaps and go from low to high frequency*

Description

step_ts_pad creates a a *specification* of a recipe step that will analyze a Date or Date-time column adding rows at a specified interval.

Usage

```
step_ts_pad(
  recipe,
  ...,
  by = "day",
  pad_value = NA,
  role = "predictor",
  trained = FALSE,
  columns = NULL,
  skip = FALSE,
  id = rand_id("ts_padding")
)

## S3 method for class 'step_ts_pad'
tidy(x, ...)
```

Arguments

recipe	A recipe object. The step will be added to the sequence of operations for this recipe.
...	A single column with class Date or POSIXct. See recipes::selections() for more details. For the tidy method, these are not currently used.
by	Either "auto", a time-based frequency like "year", "month", "day", "hour", etc, or a time expression like "5 min", or "7 days". See Details.
pad_value	Fills in padded values. Default is NA.
role	For model terms created by this step, what analysis role should they be assigned?. By default, the function assumes that the new variable columns created by the original variables will be used as predictors in a model.
trained	A logical to indicate if the quantities for preprocessing have been estimated.
columns	A character string of variables that will be used as inputs. This field is a placeholder and will be populated once recipes::prep() is used.
skip	A logical. Should the step be skipped when the recipe is baked by bake.recipe()? While all operations are baked when prep.recipe() is run, some operations may not be able to be conducted on new data (e.g. processing the outcome variable(s)). Care should be taken when using skip = TRUE as it may affect the computations for subsequent operations.

id	A character string that is unique to this step to identify it.
x	A step_ts_pad object.

Details

Date Variable

- Only one date or date-time variable may be supplied.
- step_ts_pad()) does *not* remove the original date variables.

Interval Specification (by)

Padding can be applied in the following ways:

- The eight intervals in are: year, quarter, month, week, day, hour, min, and sec.
- Intervals like 30 minutes, 1 hours, 14 days are possible.

Imputing Missing Values

The generic pad_value defaults to NA, which typically requires imputation. Some common strategies include:

- **Numeric data:** The step_ts_impute() preprocessing step can be used to impute numeric time series data with or without seasonality
- **Nominal data:** The step_mode_impute() preprocessing step can be used to replace missing values with the most common value.

Value

For step_ts_pad, an updated version of recipe with the new step added to the sequence of existing steps (if any). For the tidy method, a tibble with columns terms (the selectors or variables selected), value (the feature names).

See Also

Padding & Imputation:

- Pad Time Series: step_ts_pad()
- Impute missing values with these: step_ts_impute(), step_ts_clean()

Time Series Analysis:

- Engineered Features: step_timeseries_signature(), step_holiday_signature(), step_fourier()
- Diffs & Lags step_diff(), recipes::step_lag()
- Smoothing: step_slidify(), step_smooth()
- Variance Reduction: step_box_cox()

Main Recipe Functions:

- recipes::recipe()
- recipes::prep
- recipes::bake

Examples

```
library(recipes)
library(dplyr)

FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)


rec_obj <- recipe(adjusted ~ ., data = FB_tbl) %>%
    step_ts_pad(date, by = "day", pad_value = NA)

# View the recipe object
rec_obj

# Prepare the recipe object
prep(rec_obj)

# Bake the recipe object - Adds the padding
bake(prep(rec_obj), FB_tbl)

# Tidy shows which features have been added during the 1st step
#  in this case, step 1 is the step_timeseries_signature step
tidy(prep(rec_obj))
tidy(prep(rec_obj), number = 1)
```

---

summarise_by_time *Summarise (for Time Series Data)*

Description

summarise_by_time() is a time-based variant of the popular dplyr::summarise() function that uses .date_var to specify a date or date-time column and .by to group the calculation by groups like "5 seconds", "week", or "3 months".

summarise_by_time() and summarize_by_time() are synonyms.

Usage

```
summarise_by_time(
  .data,
  .date_var,
  .by = "day",
  ...,
  .type = c("floor", "ceiling", "round"),
  .week_start = NULL
)
```

```
summarize_by_time(
  .data,
  .date_var,
  .by = "day",
  ...,
  .type = c("floor", "ceiling", "round"),
  .week_start = NULL
)
```

Arguments

`.data` A `tbl` object or `data.frame`

`.date_var` A column containing date or date-time values to summarize. If missing, attempts to auto-detect date column.

`.by` A time unit to summarise by. Time units are collapsed using `lubridate::floor_date()` or `lubridate::ceiling_date()`.

The value can be:

- `second`
- `minute`
- `hour`
- `day`
- `week`
- `month`
- `bimonth`
- `quarter`
- `season`
- `halfyear`
- `year`

Arbitrary unique English abbreviations as in the `lubridate::period()` constructor are allowed.

`...` Name-value pairs of summary functions. The name will be the name of the variable in the result.

The value can be:

- A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`.
- A vector of length `n`, e.g. `quantile()`.
- A data frame, to add multiple columns from a single expression.

`.type` One of "floor", "ceiling", or "round. Defaults to "floor". See `lubridate::round_date`.

`.week_start` when unit is weeks, specify the reference day. 7 represents Sunday and 1 represents Monday.

Value

A `tibble` or `data.frame`

Useful summary functions

- Sum: sum()
- Center: mean(), median()
- Spread: sd(), var()
- Range: min(), max()
- Count: dplyr::n(), dplyr::n_distinct()
- Position: dplyr::first(), dplyr::last(), dplyr::nth()
- Correlation: cor(), cov()

See Also

Time-Based dplyr functions:

- summarise_by_time() - Easily summarise using a date column.
- mutate_by_time() - Simplifies applying mutations by time windows.
- filter_by_time() - Quickly filter using date ranges.
- filter_period() - Apply filtering expressions inside periods (windows)
- between_time() - Range detection for date or date-time sequences.
- pad_by_time() - Insert time series rows with regularly spaced timestamps
- condense_period() - Convert to a different periodicity
- slidify() - Turn any function into a sliding (rolling) function

Examples

```
# Libraries
library(dplyr)

# First value in each month
m4_daily %>%
    group_by(id) %>%
    summarise_by_time(
        .date_var = date,
        .by       = "month", # Setup for monthly aggregation
        # Summarization
        value  = first(value)
    )

# Last value in each month (day is first day of next month with ceiling option)
m4_daily %>%
    group_by(id) %>%
    summarise_by_time(
        .by        = "month",
        value      = last(value),
        .type      = "ceiling"
    ) %>%
    # Shift to the last day of the month
    mutate(date = date %-time% "1 day")
```

```
# Total each year (.by is set to "year" now)
m4_daily %>%
    group_by(id) %>%
    summarise_by_time(
        .by        = "year",
        value      = sum(value)
    )
```

`taylor_30_min` *Half-hourly electricity demand*

Description

Half-hourly electricity demand in England and Wales from Monday 5 June 2000 to Sunday 27 August 2000. Discussed in Taylor (2003).

Usage

```
taylor_30_min
```

Format

A tibble: 4,032 x 2

- `date`: A date-time variable in 30-minute increments
- `value`: Electricity demand in Megawatts

Source

James W Taylor

References

Taylor, J.W. (2003) Short-term electricity demand forecasting using double seasonal exponential smoothing. *Journal of the Operational Research Society*, **54**, 799-805.

Examples

```
taylor_30_min
```

time_arithmetic *Add / Subtract (For Time Series)*

Description

The easiest way to add / subtract a period to a time series date or date-time vector.

Usage

```
add_time(index, period)

subtract_time(index, period)

index %+time% period

index %-time% period
```

Arguments

index	A date or date-time vector. Can also accept a character representation.
period	A period to add. Accepts character strings like "5 seconds", "2 days", and complex strings like "1 month 4 days 34 minutes".

Details

A convenient wrapper for `lubridate::period()`. Adds and subtracts a period from a time-based index. Great for:

- Finding a timestamp n-periods into the future or past
- Shifting a time-based index. Note that `NA` values may be present where dates don't exist.

Period Specification

The `period` argument accepts complex strings like:

- "1 month 4 days 43 minutes"
- "second = 3, minute = 1, hour = 2, day = 13, week = 1"

Value

A `date` or datetime (`POSIXct`) vector the same length as `index` with the time values shifted +/- a period.

See Also

Other Time-Based vector functions:

- `between_time()` - Range detection for date or date-time sequences.

Underlying function:

- `lubridate::period()`

Examples

```
# ---- LOCATING A DATE N-PERIODS IN FUTURE / PAST ----

# Forward (Plus Time)
"2021" %+time% "1 hour 34 seconds"
"2021" %+time% "3 months"
"2021" %+time% "1 year 3 months 6 days"

# Backward (Minus Time)
"2021" %-time% "1 hour 34 seconds"
"2021" %-time% "3 months"
"2021" %-time% "1 year 3 months 6 days"

# ---- INDEX SHIFTING ----

index_daily <- tk_make_timeseries("2016", "2016-02-01")

# ADD TIME
# - Note `NA` values created where a daily dates aren't possible
#   (e.g. Feb 29 & 30, 2016 doesn't exist).
index_daily %+time% "1 month"

# Subtracting Time
index_daily %-time% "1 month"
```

`time_series_cv` *Time Series Cross Validation*

Description

Create `rsample` cross validation sets for time series. This function produces a sampling plan starting with the most recent time series observations, rolling backwards. The sampling procedure is similar to `rsample::rolling_origin()`, but places the focus of the cross validation on the most recent time series data.

Usage

```
time_series_cv(
  data,
  date_var = NULL,
  initial = 5,
  assess = 1,
  skip = 1,
  lag = 0,
```

```
  cumulative = FALSE,
  slice_limit = n(),
  point_forecast = FALSE,
  ...
)
```

Arguments

data	A data frame.
date_var	A date or date-time variable.
initial	The number of samples used for analysis/modeling in the initial resample.
assess	The number of samples used for each assessment resample.
skip	A integer indicating how many (if any) *additional* resamples to skip to thin the total amount of data points in the analysis resample. See the example below.
lag	A value to include an lag between the assessment and analysis set. This is useful if lagged predictors will be used during training and testing.
cumulative	A logical. Should the analysis resample grow beyond the size specified by `initial` at each resample?.
slice_limit	The number of slices to return. Set to `dplyr::n()`, which returns the maximum number of slices.
point_forecast	Whether or not to have the testing set be a single point forecast or to be a forecast horizon. The default is to be a forecast horizon. Default: `FALSE`
...	These dots are for future extensions and must be empty.

Details

Time-Based Specification

The `initial`, `assess`, `skip`, and `lag` variables can be specified as:

- Numeric: `initial = 24`
- Time-Based Phrases: `initial = "2 years"`, if the `data` contains a `date_var` (date or date-time)

Initial (Training Set) and Assess (Testing Set)

The main options, `initial` and `assess`, control the number of data points from the original data that are in the analysis (training set) and the assessment (testing set), respectively.

Skip

`skip` enables the function to not use every data point in the resamples. When `skip = 1`, the resampling data sets will increment by one position.

Example: Suppose that the rows of a data set are consecutive days. Using `skip = 7` will make the analysis data set operate on *weeks* instead of days. The assessment set size is not affected by this option.

Lag

The Lag parameter creates an overlap between the Testing set. This is needed when lagged predictors are used.

Cumulative vs Sliding Window

When cumulative = TRUE, the initial parameter is ignored and the analysis (training) set will grow as resampling continues while the assessment (testing) set size will always remain static.

When cumulative = FALSE, the initial parameter fixes the analysis (training) set and resampling occurs over a fixed window.

Slice Limit

This controls the number of slices. If slice_limit = 5, only the most recent 5 slices will be returned.

Point Forecast

A point forecast is sometimes desired such as if you want to forecast a value "4-weeks" into the future. You can do this by setting the following parameters:

- assess = "4 weeks"
- point_forecast = TRUE

Panel Data / Time Series Groups / Overlapping Timestamps

Overlapping timestamps occur when your data has more than one time series group. This is sometimes called *Panel Data* or *Time Series Groups*.

When timestamps are duplicated (as in the case of "Panel Data" or "Time Series Groups"), the resample calculation applies a sliding window of fixed length to the dataset. See the example using walmart_sales_weekly dataset below.

Value

An tibble with classes time_series_cv, rset, tbl_df, tbl, and data.frame. The results include a column for the data split objects and a column called id that has a character string with the resample identifier.

See Also

- time_series_cv() and rsample::rolling_origin() - Functions used to create time series resample specifications.
- plot_time_series_cv_plan() - The plotting function used for visualizing the time series resample plan.
- time_series_split() - A convenience function to return a single time series split containing a training/testing sample.

Examples

```
library(dplyr)

# DATA ----
m750 <- m4_monthly %>% dplyr::filter(id == "M750")


# RESAMPLE SPEC ----
resample_spec <- time_series_cv(data = m750,
```

```
                                    initial     = "6 years",
                                    assess      = "24 months",
                                    skip        = "24 months",
                                    cumulative  = FALSE,
                                    slice_limit = 3)

resample_spec

# VISUALIZE CV PLAN ----

# Select date and value columns from the tscv diagnostic tool
resample_spec %>% tk_time_series_cv_plan()

# Plot the date and value columns to see the CV Plan
resample_spec %>%
    plot_time_series_cv_plan(date, value, .interactive = FALSE)


# PANEL DATA / TIME SERIES GROUPS ----
# - Time Series Groups are processed using an *ungrouped* data set
# - The data has sliding windows applied starting with the beginning of the series
# - The seven groups of weekly time series are
#   processed together for <split [358/78]> dimensions

walmart_tscv <- walmart_sales_weekly %>%
    time_series_cv(
        date_var    = Date,
        initial     = "12 months",
        assess      = "3 months",
        skip        = "3 months",
        slice_limit = 4
    )

walmart_tscv

walmart_tscv %>%
    plot_time_series_cv_plan(Date, Weekly_Sales, .interactive = FALSE)
```

time_series_split *Simple Training/Test Set Splitting for Time Series*

Description

time_series_split creates resample splits using time_series_cv() but returns only a **single split.** This is useful when creating a single train/test split.

Usage

```
time_series_split(
```

```
  data,
  date_var = NULL,
  initial = 5,
  assess = 1,
  skip = 1,
  lag = 0,
  cumulative = FALSE,
  slice = 1,
  point_forecast = FALSE,
  ...
)
```

Arguments

`data`	A data frame.
`date_var`	A date or date-time variable.
`initial`	The number of samples used for analysis/modeling in the initial resample.
`assess`	The number of samples used for each assessment resample.
`skip`	A integer indicating how many (if any) *additional* resamples to skip to thin the total amount of data points in the analysis resample. See the example below.
`lag`	A value to include an lag between the assessment and analysis set. This is useful if lagged predictors will be used during training and testing.
`cumulative`	A logical. Should the analysis resample grow beyond the size specified by `initial` at each resample?.
`slice`	Returns a single slice from time_series_cv
`point_forecast`	Whether or not to have the testing set be a single point forecast or to be a forecast horizon. The default is to be a forecast horizon. Default: `FALSE`
`...`	These dots are for future extensions and must be empty.

Details

Time-Based Specification

The `initial`, `assess`, `skip`, and `lag` variables can be specified as:

- Numeric: `initial = 24`
- Time-Based Phrases: `initial = "2 years"`, if the `data` contains a `date_var` (date or date-time)

Initial (Training Set) and Assess (Testing Set)

The main options, `initial` and `assess`, control the number of data points from the original data that are in the analysis (training set) and the assessment (testing set), respectively.

Skip

`skip` enables the function to not use every data point in the resamples. When `skip = 1`, the resampling data sets will increment by one position.

Example: Suppose that the rows of a data set are consecutive days. Using `skip = 7` will make the analysis data set operate on *weeks* instead of days. The assessment set size is not affected by this option.

Lag

The Lag parameter creates an overlap between the Testing set. This is needed when lagged predictors are used.

Cumulative vs Sliding Window

When `cumulative = TRUE`, the `initial` parameter is ignored and the analysis (training) set will grow as resampling continues while the assessment (testing) set size will always remain static.

When `cumulative = FALSE`, the `initial` parameter fixes the analysis (training) set and resampling occurs over a fixed window.

Slice

This controls which slice is returned. If `slice = 1`, only the most recent slice will be returned.

Value

An `rsplit` object that can be used with the `training` and `testing` functions to extract the data in each split.

See Also

- `time_series_cv()` and `rsample::rolling_origin()` - Functions used to create time series resample specifications.

Examples

```
library(dplyr)

# DATA ----
m750 <- m4_monthly %>% dplyr::filter(id == "M750")

# Get the most recent 3 years as testing, and previous 10 years as training
m750 %>%
    time_series_split(initial = "10 years", assess = "3 years")

# Skip the most recent 3 years
m750 %>%
    time_series_split(
        initial = "10 years",
        assess  = "3 years",
        skip    = "3 years",
        slice   = 2          # <- Returns 2nd slice, 3-years back
    )

# Add 1 year lag for testing overlap
m750 %>%
    time_series_split(
        initial = "10 years",
        assess  = "3 years",
```

```
    skip    = "3 years",
    slice   = 2,
    lag     = "1 year"   # <- Overlaps training/testing by 1 year
)
```

`tk_acf_diagnostics` *Group-wise ACF, PACF, and CCF Data Preparation*

Description

The `tk_acf_diagnostics()` function provides a simple interface to detect Autocorrelation (ACF), Partial Autocorrelation (PACF), and Cross Correlation (CCF) of Lagged Predictors in one `tibble`. This function powers the `plot_acf_diagnostics()` visualization.

Usage

```
tk_acf_diagnostics(.data, .date_var, .value, .ccf_vars = NULL, .lags = 1000)
```

Arguments

`.data`	A data frame or tibble with numeric features (values) in descending chronological order
`.date_var`	A column containing either date or date-time values
`.value`	A numeric column with a value to have ACF and PACF calculations performed.
`.ccf_vars`	Additional features to perform Lag Cross Correlations (CCFs) versus the `.value`. Useful for evaluating external lagged regressors.
`.lags`	A seqence of one or more lags to evaluate.

Details

Simplified ACF, PACF, & CCF

We are often interested in all 3 of these functions. Why not get all 3 at once? Now you can!

- **ACF** - Autocorrelation between a target variable and lagged versions of itself
- **PACF** - Partial Autocorrelation removes the dependence of lags on other lags highlighting key seasonalities.
- **CCF** - Shows how lagged predictors can be used for prediction of a target variable.

Lag Specification

Lags (`.lags`) can either be specified as:

- A time-based phrase indicating a duraction (e.g. `2 months`)
- A maximum lag (e.g. `.lags = 28`)
- A sequence of lags (e.g. `.lags = 7:28`)

Scales to Multiple Time Series with Groupes

The `tk_acf_diagnostics()` works with `grouped_df`'s, meaning you can group your time series by one or more categorical columns with `dplyr::group_by()` and then apply `tk_acf_diagnostics()` to return group-wise lag diagnostics.

Special Note on Dots (...)

Unlike other plotting utilities, the `...` arguments is NOT used for group-wise analysis. Rather, it's used for processing Cross Correlations (CCFs).

Use `dplyr::group_by()` for processing multiple time series groups.

Value

A `tibble` or `data.frame` containing the autocorrelation, partial autocorrelation and cross correlation data.

See Also

- **Visualizing ACF, PACF, & CCF:** `plot_acf_diagnostics()`
- **Visualizing Seasonality:** `plot_seasonal_diagnostics()`
- **Visualizing Time Series:** `plot_time_series()`

Examples

```
library(dplyr)

# ACF, PACF, & CCF in 1 Data Frame
# - Get ACF & PACF for target (adjusted)
# - Get CCF between adjusted and volume and close
FANG %>%
    filter(symbol == "FB") %>%
    tk_acf_diagnostics(date, adjusted,                # ACF & PACF
                       .ccf_vars = c(volume, close),  # CCFs
                       .lags     = 500)

# Scale with groups using group_by()
FANG %>%
    group_by(symbol) %>%
    tk_acf_diagnostics(date, adjusted,
                       .ccf_vars = c(volume, close),
                       .lags     = "3 months")

# Apply Transformations
FANG %>%
    group_by(symbol) %>%
    tk_acf_diagnostics(
        date, diff_vec(adjusted),  # Apply differencing transformation
        .lags = 0:500
    )
```

tk_anomaly_diagnostics *Automatic group-wise Anomaly Detection by STL Decomposition*

Description

tk_anomaly_diagnostics() is the preprocessor for plot_anomaly_diagnostics(). It performs automatic anomaly detection for one or more time series groups.

Usage

```
tk_anomaly_diagnostics(
  .data,
  .date_var,
  .value,
  .frequency = "auto",
  .trend = "auto",
  .alpha = 0.05,
  .max_anomalies = 0.2,
  .message = TRUE
)
```

Arguments

.data	A tibble or data.frame with a time-based column
.date_var	A column containing either date or date-time values
.value	A column containing numeric values
.frequency	Controls the seasonal adjustment (removal of seasonality). Input can be either "auto", a time-based definition (e.g. "2 weeks"), or a numeric number of observations per frequency (e.g. 10). Refer to tk_get_frequency().
.trend	Controls the trend component. For STL, trend controls the sensitivity of the LOESS smoother, which is used to remove the remainder. Refer to tk_get_trend().
.alpha	Controls the width of the "normal" range. Lower values are more conservative while higher values are less prone to incorrectly classifying "normal" observations.
.max_anomalies	The maximum percent of anomalies permitted to be identified.
.message	A boolean. If TRUE, will output information related to automatic frequency and trend selection (if applicable).

Details

The tk_anomaly_diagnostics() method for anomaly detection that implements a 2-step process to detect outliers in time series.

Step 1: Detrend & Remove Seasonality using STL Decomposition

The decomposition separates the "season" and "trend" components from the "observed" values leaving the "remainder" for anomaly detection.

The user can control two parameters: frequency and trend.

1. `.frequency`: Adjusts the "season" component that is removed from the "observed" values.
2. `.trend`: Adjusts the trend window (t.window parameter from `stats::stl()` that is used.

The user may supply both `.frequency` and `.trend` as time-based durations (e.g. "6 weeks") or numeric values (e.g. 180) or "auto", which predetermines the frequency and/or trend based on the scale of the time series using the `tk_time_scale_template()`.

Step 2: Anomaly Detection

Once "trend" and "season" (seasonality) is removed, anomaly detection is performed on the "remainder". Anomalies are identified, and boundaries (recomposed_l1 and recomposed_l2) are determined.

The Anomaly Detection Method uses an inner quartile range (IQR) of +/-25 the median.

IQR Adjustment, alpha parameter

With the default `alpha = 0.05`, the limits are established by expanding the 25/75 baseline by an IQR Factor of 3 (3X). The *IQR Factor = 0.15 / alpha* (hence 3X with alpha = 0.05):

- To increase the IQR Factor controlling the limits, decrease the alpha, which makes it more difficult to be an outlier.
- Increase alpha to make it easier to be an outlier.
- The IQR outlier detection method is used in `forecast::tsoutliers()`.
- A similar outlier detection method is used by Twitter's `AnomalyDetection` package.
- Both Twitter and Forecast tsoutliers methods have been implemented in Business Science's `anomalize` package.

Value

A `tibble` or `data.frame` with STL Decomposition Features (observed, season, trend, remainder, seasadj) and Anomaly Features (remainder_l1, remainder_l2, anomaly, recomposed_l1, and recomposed_l2)

References

1. CLEVELAND, R. B., CLEVELAND, W. S., MCRAE, J. E., AND TERPENNING, I. STL: A Seasonal-Trend Decomposition Procedure Based on Loess. Journal of Official Statistics, Vol. 6, No. 1 (1990), pp. 3-73.
2. Owen S. Vallis, Jordan Hochenbaum and Arun Kejariwal (2014). A Novel Technique for Long-Term Anomaly Detection in the Cloud. Twitter Inc.

See Also

- `plot_anomaly_diagnostics()`: Visual anomaly detection

Examples

```
library(dplyr)

walmart_sales_weekly %>%
    filter(id %in% c("1_1", "1_3")) %>%
    group_by(id) %>%
    tk_anomaly_diagnostics(Date, Weekly_Sales)
```

---

tk_augment_differences

Add many differenced columns to the data

Description

A handy function for adding multiple lagged difference values to a data frame. Works with dplyr groups too.

Usage

```
tk_augment_differences(
  .data,
  .value,
  .lags = 1,
  .differences = 1,
  .log = FALSE,
  .names = "auto"
)
```

Arguments

.data	A tibble.
.value	One or more column(s) to have a transformation applied. Usage of tidyselect functions (e.g. contains()) can be used to select multiple columns.
.lags	One or more lags for the difference(s)
.differences	The number of differences to apply.
.log	If TRUE, applies log-differences.
.names	A vector of names for the new columns. Must be of same length as the number of output columns. Use "auto" to automatically rename the columns.

Details

Benefits

This is a scalable function that is:

- Designed to work with grouped data using dplyr::group_by()
- Add multiple differences by adding a sequence of differences using the .lags argument (e.g. lags = 1:20)

Value

Returns a tibble object describing the timeseries.

See Also

Augment Operations:

- tk_augment_timeseries_signature() - Group-wise augmentation of timestamp features
- tk_augment_holiday_signature() - Group-wise augmentation of holiday features
- tk_augment_slidify() - Group-wise augmentation of rolling functions
- tk_augment_lags() - Group-wise augmentation of lagged data
- tk_augment_differences() - Group-wise augmentation of differenced data
- tk_augment_fourier() - Group-wise augmentation of fourier series

Underlying Function:

- diff_vec() - Underlying function that powers tk_augment_differences()

Examples

```
library(dplyr)

m4_monthly %>%
    group_by(id) %>%
    tk_augment_differences(value, .lags = 1:20)
```

---

tk_augment_fourier *Add many fourier series to the data*

Description

A handy function for adding multiple fourier series to a data frame. Works with dplyr groups too.

Usage

```
tk_augment_fourier(.data, .date_var, .periods, .K = 1, .names = "auto")
```

Arguments

.data	A tibble.
.date_var	A date or date-time column used to calculate a fourier series
.periods	One or more periods for the fourier series
.K	The maximum number of fourier orders.
.names	A vector of names for the new columns. Must be of same length as the number of output columns. Use "auto" to automatically rename the columns.

Details

Benefits

This is a scalable function that is:

- Designed to work with grouped data using `dplyr::group_by()`
- Add multiple differences by adding a sequence of differences using the `.periods` argument (e.g. `lags = 1:20`)

Value

Returns a `tibble` object describing the timeseries.

See Also

Augment Operations:

- `tk_augment_timeseries_signature()` - Group-wise augmentation of timestamp features
- `tk_augment_holiday_signature()` - Group-wise augmentation of holiday features
- `tk_augment_slidify()` - Group-wise augmentation of rolling functions
- `tk_augment_lags()` - Group-wise augmentation of lagged data
- `tk_augment_differences()` - Group-wise augmentation of differenced data
- `tk_augment_fourier()` - Group-wise augmentation of fourier series

Underlying Function:

- `fourier_vec()` - Underlying function that powers `tk_augment_fourier()`

Examples

```
library(dplyr)

m4_monthly %>%
    group_by(id) %>%
    tk_augment_fourier(date, .periods = c(6, 12), .K = 2)
```

`tk_augment_holiday` *Add many holiday features to the data*

Description

Quickly add the "holiday signature" - sets of holiday features that correspond to calendar dates. Works with `dplyr` groups too.

Usage

```
tk_augment_holiday_signature(
  .data,
  .date_var = NULL,
  .holiday_pattern = ".",
  .locale_set = c("all", "none", "World", "US", "CA", "GB", "FR", "IT", "JP", "CH", "DE"),
  .exchange_set = c("all", "none", "NYSE", "LONDON", "NERC", "TSX", "ZURICH")
)
```

Arguments

`.data`	A time-based tibble or time-series object.
`.date_var`	A column containing either date or date-time values. If `NULL`, the time-based column will interpret from the object (tibble).
`.holiday_pattern`	A regular expression pattern to search the "Holiday Set".
`.locale_set`	Return binary holidays based on locale. One of: "all", "none", "World", "US", "CA", "GB", "FR", "IT", "JP", "CH", "DE".
`.exchange_set`	Return binary holidays based on Stock Exchange Calendars. One of: "all", "none", "NYSE", "LONDON", "NERC", "TSX", "ZURICH".

Details

`tk_augment_holiday_signature` adds the holiday signature features. See `tk_get_holiday_signature()` (powers the augment function) for a full description and examples for how to use.

1. Individual Holidays

These are **single holiday features** that can be filtered using a pattern. This helps in identifying which holidays are important to a machine learning model. This can be useful for example in **e-commerce initiatives** (e.g. sales during Christmas and Thanskgiving).

2. Locale-Based Summary Sets

Locale-based holdiay sets are useful for **e-commerce initiatives** (e.g. sales during Christmas and Thanskgiving). Filter on a locale to identify all holidays in that locale.

3. Stock Exchange Calendar Summary Sets

Exchange-based holdiay sets are useful for identifying **non-working days.** Filter on an index to identify all holidays that are commonly non-working.

Value

Returns a `tibble` object describing the holiday timeseries.

See Also

Augment Operations:

- `tk_augment_timeseries_signature()` - Group-wise augmentation of timestamp features
- `tk_augment_holiday_signature()` - Group-wise augmentation of holiday features

- tk_augment_slidify() - Group-wise augmentation of rolling functions
- tk_augment_lags() - Group-wise augmentation of lagged data
- tk_augment_differences() - Group-wise augmentation of differenced data
- tk_augment_fourier() - Group-wise augmentation of fourier series

Underlying Function:

- tk_get_holiday_signature() - Underlying function that powers holiday feature generation

Examples

```
library(dplyr)

dates_in_2017_tbl <- tibble(index = tk_make_timeseries("2017-01-01", "2017-12-31", by = "day"))

# Non-working days in US due to Holidays using NYSE stock exchange calendar
dates_in_2017_tbl %>%
    tk_augment_holiday_signature(
        index,
        .holiday_pattern = "^$",   # Returns nothing on purpose
        .locale_set      = "none",
        .exchange_set    = "NYSE")

# All holidays in US
dates_in_2017_tbl %>%
    tk_augment_holiday_signature(
        index,
        .holiday_pattern = "US_",
        .locale_set      = "US",
        .exchange_set    = "none")

# All holidays for World and Italy-specific Holidays
# - Note that Italy celebrates specific holidays in addition to many World Holidays
dates_in_2017_tbl %>%
    tk_augment_holiday_signature(
        index,
        .holiday_pattern = "(World)|(IT_)",
        .locale_set      = c("World", "IT"),
        .exchange_set    = "none")
```

---

tk_augment_lags *Add many lags to the data*

Description

A handy function for adding multiple lagged columns to a data frame. Works with dplyr groups too.

Usage

```
tk_augment_lags(.data, .value, .lags = 1, .names = "auto")

tk_augment_leads(.data, .value, .lags = -1, .names = "auto")
```

Arguments

.data	A tibble.
.value	One or more column(s) to have a transformation applied. Usage of `tidyselect` functions (e.g. `contains()`) can be used to select multiple columns.
.lags	One or more lags for the difference(s)
.names	A vector of names for the new columns. Must be of same length as `.lags`.

Details

Lags vs Leads

A *negative lag* is considered a lead. The `tk_augment_leads()` function is identical to `tk_augment_lags()` with the exception that the automatic naming convetion (`.names = 'auto'`) will convert column names with negative lags to leads.

Benefits

This is a scalable function that is:

- Designed to work with grouped data using `dplyr::group_by()`
- Add multiple lags by adding a sequence of lags using the `.lags` argument (e.g. `.lags = 1:20`)

Value

Returns a `tibble` object describing the timeseries.

See Also

Augment Operations:

- `tk_augment_timeseries_signature()` - Group-wise augmentation of timestamp features
- `tk_augment_holiday_signature()` - Group-wise augmentation of holiday features
- `tk_augment_slidify()` - Group-wise augmentation of rolling functions
- `tk_augment_lags()` - Group-wise augmentation of lagged data
- `tk_augment_differences()` - Group-wise augmentation of differenced data
- `tk_augment_fourier()` - Group-wise augmentation of fourier series

Underlying Function:

- `lag_vec()` - Underlying function that powers `tk_augment_lags()`

Examples

```
library(dplyr)

# Lags
m4_monthly %>%
    group_by(id) %>%
    tk_augment_lags(contains("value"), .lags = 1:20)

# Leads
m4_monthly %>%
    group_by(id) %>%
    tk_augment_leads(value, .lags = 1:-20)
```

---

tk_augment_slidify — *Add many rolling window calculations to the data*

Description

Quickly use any function as a rolling function and apply to multiple `.periods`. Works with `dplyr` groups too.

Usage

```
tk_augment_slidify(
  .data,
  .value,
  .period,
  .f,
  ...,
  .align = c("center", "left", "right"),
  .partial = FALSE,
  .names = "auto"
)
```

Arguments

`.data`	A tibble.
`.value`	One or more column(s) to have a transformation applied. Usage of `tidyselect` functions (e.g. `contains()`) can be used to select multiple columns.
`.period`	One or more periods for the rolling window(s)
`.f`	A summary `[function / formula]`,
`...`	Optional arguments for the summary function
`.align`	Rolling functions generate `.period - 1` fewer values than the incoming vector. Thus, the vector needs to be aligned. Select one of "center", "left", or "right".

.partial — .partial Should the moving window be allowed to return partial (incomplete) windows instead of NA values. Set to FALSE by default, but can be switched to TRUE to remove NA's.

.names — A vector of names for the new columns. Must be of same length as .period. Default is "auto".

Details

tk_augment_slidify() scales the slidify_vec() function to multiple time series .periods. See slidify_vec() for examples and usage of the core function arguments.

Value

Returns a tibble object describing the timeseries.

See Also

Augment Operations:

- tk_augment_timeseries_signature() - Group-wise augmentation of timestamp features
- tk_augment_holiday_signature() - Group-wise augmentation of holiday features
- tk_augment_slidify() - Group-wise augmentation of rolling functions
- tk_augment_lags() - Group-wise augmentation of lagged data
- tk_augment_differences() - Group-wise augmentation of differenced data
- tk_augment_fourier() - Group-wise augmentation of fourier series

Underlying Function:

- slidify_vec() - The underlying function that powers tk_augment_slidify()

Examples

```
library(dplyr)

# Single Column | Multiple Rolling Windows
FANG %>%
    select(symbol, date, adjusted) %>%
    group_by(symbol) %>%
    tk_augment_slidify(
        .value   = contains("adjust"),
        # Multiple rolling windows
        .period  = c(10, 30, 60, 90),
        .f       = mean,
        .partial = TRUE,
        .names   = stringr::str_c("MA_", c(10, 30, 60, 90))
    ) %>%
    ungroup()

# Multiple Columns | Multiple Rolling Windows
FANG %>%
```

```
select(symbol, date, adjusted, volume) %>%
group_by(symbol) %>%
tk_augment_slidify(
    .value  = c(adjusted, volume),
    .period  = c(10, 30, 60, 90),
    .f       = mean,
    .partial = TRUE
) %>%
ungroup()
```

---

tk_augment_timeseries *Add many time series features to the data*

Description

Add many time series features to the data

Usage

```
tk_augment_timeseries_signature(.data, .date_var = NULL)
```

Arguments

.data	A time-based tibble or time-series object.
.date_var	For tibbles, a column containing either date or date-time values. If NULL, the time-based column will interpret from the object (tibble, xts, zoo, etc).

Details

`tk_augment_timeseries_signature()` adds 25+ time series features including:

- **Trend in Seconds Granularity:** index.num
- **Yearly Seasonality:** Year, Month, Quarter
- **Weekly Seasonality:** Week of Month, Day of Month, Day of Week, and more
- **Daily Seasonality:** Hour, Minute, Second
- **Weekly Cyclic Patterns:** 2 weeks, 3 weeks, 4 weeks

Value

Returns a `tibble` object describing the timeseries.

See Also

Augment Operations:

- `tk_augment_timeseries_signature()` - Group-wise augmentation of timestamp features
- `tk_augment_holiday_signature()` - Group-wise augmentation of holiday features
- `tk_augment_slidify()` - Group-wise augmentation of rolling functions
- `tk_augment_lags()` - Group-wise augmentation of lagged data
- `tk_augment_differences()` - Group-wise augmentation of differenced data
- `tk_augment_fourier()` - Group-wise augmentation of fourier series

Underlying Function:

- `tk_get_timeseries_signature()` - Returns timeseries features from an index

Examples

```
library(dplyr)

m4_daily %>%
    group_by(id) %>%
    tk_augment_timeseries_signature(date)
```

`tk_get_frequency` *Automatic frequency and trend calculation from a time series index*

Description

Automatic frequency and trend calculation from a time series index

Usage

```
tk_get_frequency(idx, period = "auto", message = TRUE)

tk_get_trend(idx, period = "auto", message = TRUE)
```

Arguments

idx	A date or datetime index.
period	Either "auto", a time-based definition (e.g. "2 weeks"), or a numeric number of observations per frequency (e.g. 10).
message	A boolean. If `message = TRUE`, the frequency or trend is output as a message along with the units in the scale of the data.

Details

A *frequency* is loosely defined as the number of observations that comprise a cycle in a data set. The *trend* is loosely defined as time span that can be aggregated across to visualize the central tendency of the data. It's often easiest to think of frequency and trend in terms of the time-based units that the data is already in. **This is what `tk_get_frequency()` and `time_trend()` enable: using time-based periods to define the frequency or trend.**

Frequency:

As an example, a weekly cycle is often 5-days (for working days) or 7-days (for calendar days). Rather than specify a frequency of 5 or 7, the user can specify `period = "1 week"`, and `tk_get_frequency()` will detect the scale of the time series and return 5 or 7 based on the actual data.

The `period` argument has three basic options for returning a frequency. Options include:

- `"auto"`: A target frequency is determined using a pre-defined template (see `template` below).
- `time-based duration`: (e.g. "1 week" or "2 quarters" per cycle)
- `numeric number of observations`: (e.g. 5 for 5 observations per cycle)

When `period = "auto"`, the `tk_time_scale_template()` is used to calculate the frequency.

Trend:

As an example, the trend of daily data is often best aggregated by evaluating the moving average over a quarter or a month span. Rather than specify the number of days in a quarter or month, the user can specify "1 quarter" or "1 month", and the `time_trend()` function will return the correct number of observations per trend cycle. In addition, there is an option, `period = "auto"`, to auto-detect an appropriate trend span depending on the data. The `template` is used to define the appropriate trend span.

Time Scale Template

The `tk_time_scale_template()` is a Look-Up Table used by the trend and period to find the appropriate time scale. It contains three features: `time_scale`, `frequency`, and `trend`.

The algorithm will inspect the scale of the time series and select the best frequency or trend that matches the scale and number of observations per target frequency. A frequency is then chosen on be the best match.

The predefined template is stored in a function `tk_time_scale_template()`. You can modify the template with `set_tk_time_scale_template()`.

Value

Returns a scalar numeric value indicating the number of observations in the frequency or trend span.

See Also

- Time Scale Template Modifiers: `get_tk_time_scale_template()`, `set_tk_time_scale_template()`

Examples

```
library(dplyr)

idx_FB <- FANG %>%
```

```
    filter(symbol == "FB") %>%
    pull(date)

# Automated Frequency Calculation
tk_get_frequency(idx_FB, period = "auto")

# Automated Trend Calculation
tk_get_trend(idx_FB, period = "auto")

# Manually Override Trend
tk_get_trend(idx_FB, period = "1 year")
```

---

tk_get_holiday *Get holiday features from a time-series index*

Description

Get holiday features from a time-series index

Usage

```
tk_get_holiday_signature(
  idx,
  holiday_pattern = ".",
  locale_set = c("all", "none", "World", "US", "CA", "GB", "FR", "IT", "JP", "CH", "DE"),
  exchange_set = c("all", "none", "NYSE", "LONDON", "NERC", "TSX", "ZURICH")
)

tk_get_holidays_by_year(years = year(today()))
```

Arguments

idx	A time-series index that is a vector of dates or datetimes.
holiday_pattern	A regular expression pattern to search the "Holiday Set".
locale_set	Return binary holidays based on locale. One of: "all", "none", "World", "US", "CA", "GB", "FR", "IT", "JP", "CH", "DE".
exchange_set	Return binary holidays based on Stock Exchange Calendars. One of: "all", "none", "NYSE", "LONDON", "NERC", "TSX", "ZURICH".
years	One or more years to collect holidays for.

Details

Feature engineering holidays can help identify critical patterns for machine learning algorithms. tk_get_holiday_signature() helps by providing feature sets for 3 types of features:

1. Individual Holidays

These are **single holiday features** that can be filtered using a pattern. This helps in identifying which holidays are important to a machine learning model. This can be useful for example in **e-commerce initiatives** (e.g. sales during Christmas and Thanskgiving).

2. Locale-Based Summary Sets

Locale-based holdiay sets are useful for **e-commerce initiatives** (e.g. sales during Christmas and Thanskgiving). Filter on a locale to identify all holidays in that locale.

3. Stock Exchange Calendar Summary Sets

Exchange-based holdiay sets are useful for identifying **non-working days.** Filter on an index to identify all holidays that are commonly non-working.

Value

Returns a `tibble` object describing the timeseries holidays.

See Also

- `tk_augment_holiday_signature()` - A quick way to add holiday features to a data.frame
- `step_holiday_signature()` - Preprocessing and feature engineering steps for use with `recipes`

Examples

```
library(dplyr)
library(stringr)

# Works with time-based tibbles
idx <- tk_make_timeseries("2017-01-01", "2017-12-31", by = "day")

# --- BASIC USAGE ----

tk_get_holiday_signature(idx)

# ---- FILTERING WITH PATTERNS & SETS ----

# List available holidays - see patterns
tk_get_holidays_by_year(2020) %>%
    filter(holiday_name %>% str_detect("US_"))

# Filter using holiday patterns
# - Get New Years, Christmas and Thanksgiving Features in US
tk_get_holiday_signature(
    idx,
    holiday_pattern = "(US_NewYears)|(US_Christmas)|(US_Thanks)",
    locale_set      = "none",
    exchange_set    = "none")

# ---- APPLYING FILTERS ----

# Filter with locale sets - Signals all holidays in a locale
tk_get_holiday_signature(
    idx,
```

```
    holiday_pattern = "$^", # Matches nothing on purpose
    locale_set      = "US",
    exchange_set    = "none")

# Filter with exchange sets - Signals Common Non-Business Days
tk_get_holiday_signature(
    idx,
    holiday_pattern = "$^", # Matches nothing on purpose
    locale_set      = "none",
    exchange_set    = "NYSE")
```

`tk_get_timeseries` *Get date features from a time-series index*

Description

Get date features from a time-series index

Usage

```
tk_get_timeseries_signature(idx)

tk_get_timeseries_summary(idx)
```

Arguments

`idx`	A time-series index that is a vector of dates or datetimes.

Details

`tk_get_timeseries_signature` decomposes the timeseries into commonly needed features such as numeric value, differences, year, month, day, day of week, day of month, day of year, hour, minute, second.

`tk_get_timeseries_summary` returns the summary returns the start, end, units, scale, and a "summary" of the timeseries differences in seconds including the minimum, 1st quartile, median, mean, 3rd quartile, and maximum frequency. The timeseries differences give the user a better picture of the index frequency so the user can understand the level of regularity or irregularity. A perfectly regular time series will have equal values in seconds for each metric. However, this is not often the case.

Important Note: These functions only work with time-based indexes in datetime, date, yearmon, and yearqtr values. Regularized dates cannot be decomposed.

Value

Returns a `tibble` object describing the timeseries.

See Also

tk_index(), tk_augment_timeseries_signature(), tk_make_future_timeseries()

Examples

```
library(dplyr)
library(lubridate)
library(zoo)

# Works with time-based tibbles
FB_tbl <- FANG %>% dplyr::filter(symbol == "FB")
FB_idx <- tk_index(FB_tbl)

tk_get_timeseries_signature(FB_idx)
tk_get_timeseries_summary(FB_idx)


# Works with dates in any periodicity
idx_weekly <- seq.Date(from = lubridate::ymd("2016-01-01"), by = 'week', length.out = 6)

tk_get_timeseries_signature(idx_weekly)
tk_get_timeseries_summary(idx_weekly)


# Works with zoo yearmon and yearqtr classes
idx_yearmon <- seq.Date(from       = lubridate::ymd("2016-01-01"),
                        by         = "month",
                        length.out = 12) %>%
    zoo::as.yearmon()

tk_get_timeseries_signature(idx_yearmon)
tk_get_timeseries_summary(idx_yearmon)
```

`tk_get_timeseries_unit_frequency`
Get the timeseries unit frequency for the primary time scales

---

Description

Get the timeseries unit frequency for the primary time scales

Usage

```
tk_get_timeseries_unit_frequency()
```

Value

`tk_get_timeseries_unit_frequency` returns a tibble containing the timeseries frequencies in seconds for the primary time scales including "sec", "min", "hour", "day", "week", "month", "quarter", and "year".

Examples

```
tk_get_timeseries_unit_frequency()
```

---

tk_get_timeseries_variables

Get date or datetime variables (column names)

Description

Get date or datetime variables (column names)

Usage

```
tk_get_timeseries_variables(data)
```

Arguments

data	An object of class `data.frame`

Details

`tk_get_timeseries_variables` returns the column names of date or datetime variables in a data frame. Classes that meet criteria for return include those that inherit `POSIXt`, `Date`, `zoo::yearmon`, `zoo::yearqtr`. Function was adapted from `padr:::get_date_variables()`. See padr helpers.R

Value

`tk_get_timeseries_variables` returns a vector containing column names of date-like classes.

Examples

```
library(dplyr)

FANG %>%
    tk_get_timeseries_variables()
```

tk_index *Extract an index of date or datetime from time series objects, models, forecasts*

Description

Extract an index of date or datetime from time series objects, models, forecasts

Usage

```
tk_index(data, timetk_idx = FALSE, silent = FALSE)

has_timetk_idx(data)
```

Arguments

data	A time-based tibble, time-series object, time-series model, or forecast object.
timetk_idx	If timetk_idx is TRUE a timetk time-based index attribute is attempted to be returned. If FALSE the default index is returned. See discussion below for further details.
silent	Used to toggle printing of messages and warnings.

Details

tk_index() is used to extract the date or datetime index from various time series objects, models and forecasts. The method can be used on tbl, xts, zoo, zooreg, and ts objects. The method can additionally be used on forecast objects and a number of objects generated by modeling functions such as Arima, ets, and HoltWinters classes to get the index of the underlying data.

The boolean timetk_idx argument is applicable to regularized time series objects such as ts and zooreg classes that have both a regularized index and *potentially* a "timetk index" (a time-based attribute). When set to FALSE the regularized index is returned. When set to TRUE the time-based timetk index is returned *if present*.

has_timetk_idx() is used to determine if the object has a "timetk index" attribute and can thus benefit from the tk_index(timetk_idx = TRUE). TRUE indicates the "timetk index" attribute is present. FALSE indicates the "timetk index" attribute is not present. If FALSE, the tk_index() function will return the default index for the data type.

Important Note: To gain the benefit of timetk_idx the time series must have a timetk index. Use has_timetk_idx to determine if the object has a timetk index. This is particularly important for ts objects, which by default do not contain a time-based index and therefore must be coerced from time-based objects such as tbl, xts, or zoo using the tk_ts() function in order to get the "timetk index" attribute. Refer to tk_ts() for creating persistent date / datetime index during coercion to ts.

Value

Returns a vector of date or date times

See Also

tk_ts(), tk_tbl(), tk_xts(), tk_zoo(), tk_zooreg()

Examples

```
# Create time-based tibble
data_tbl <- tibble::tibble(
    date = seq.Date(from = as.Date("2000-01-01"), by = 1, length.out = 5),
    x    = rnorm(5) * 10,
    y    = 5:1
)
tk_index(data_tbl) # Returns time-based index vector

# Coerce to ts using tk_ts(): Preserves time-basis
data_ts <- tk_ts(data_tbl)
tk_index(data_ts, timetk_idx = FALSE) # Returns regularized index
tk_index(data_ts, timetk_idx = TRUE)  # Returns original time-based index vector

# Coercing back to tbl
tk_tbl(data_ts, timetk_idx = FALSE) # Returns regularized tbl
tk_tbl(data_ts, timetk_idx = TRUE)  # Returns time-based tbl
```

---

tk_make_future_timeseries *Make future time series from existing*

Description

Make future time series from existing

Usage

```
tk_make_future_timeseries(
  idx,
  length_out,
  inspect_weekdays = FALSE,
  inspect_months = FALSE,
  skip_values = NULL,
  insert_values = NULL,
  n_future = NULL
)
```

Arguments

- `idx` A vector of dates
- `length_out` Number of future observations. Can be numeric number or a phrase like "1 year".
- `inspect_weekdays` Uses a logistic regression algorithm to inspect whether certain weekdays (e.g. weekends) should be excluded from the future dates. Default is `FALSE`.
- `inspect_months` Uses a logistic regression algorithm to inspect whether certain days of months (e.g. last two weeks of year or seasonal days) should be excluded from the future dates. Default is `FALSE`.
- `skip_values` A vector of same class as `idx` of timeseries values to skip.
- `insert_values` A vector of same class as `idx` of timeseries values to insert.
- `n_future` (DEPRECATED) Number of future observations. Can be numeric number or a phrase like "1 year".

Details

Future Sequences

`tk_make_future_timeseries` returns a time series based on the input index frequency and attributes.

Specifying Length of Future Observations

The argument `length_out` determines how many future index observations to compute. It can be specified as:

- **A numeric value** - the number of future observations to return.
  - The number of observations returned is *always* equal to the value the user inputs.
  - The **end date can vary** based on the number of timestamps chosen.
- **A time-based phrase** - The duration into the future to include (e.g. "6 months" or "30 minutes").
  - The *duration* defines the *end date* for observations.
  - The **end date will not change** and those timestamps that fall within the end date will be returned (e.g. a quarterly time series will return 4 quarters if `length_out = "1 year"`).
  - The number of observations will vary to fit within the end date.

Weekday and Month Inspection

The `inspect_weekdays` and `inspect_months` arguments apply to "daily" (scale = "day") data (refer to `tk_get_timeseries_summary()` to get the index scale).

- The `inspect_weekdays` argument is useful in determining missing days of the week that occur on a weekly frequency such as every week, every other week, and so on. It's recommended to have at least 60 days to use this option.
- The `inspect_months` argument is useful in determining missing days of the month, quarter or year; however, the algorithm can inadvertently select incorrect dates if the pattern is erratic.

Skipping / Inserting Values

The skip_values and insert_values arguments can be used to remove and add values into the series of future times. The values must be the same format as the idx class.

- The skip_values argument useful for passing holidays or special index values that should be excluded from the future time series.
- The insert_values argument is useful for adding values back that the algorithm may have excluded.

Value

A vector containing future index of the same class as the incoming index idx

See Also

- Making Time Series: tk_make_timeseries()
- Working with Holidays & Weekends: tk_make_holiday_sequence(), tk_make_weekend_sequence(), tk_make_weekday_sequence()
- Working with Timestamp Index: tk_index(), tk_get_timeseries_summary(), tk_get_timeseries_signature()

Examples

```
library(dplyr)

# Basic example - By 3 seconds
idx <- tk_make_timeseries("2016-01-01 00:00:00", by = "3 sec", length_out = 3)
idx

# Make next three timestamps in series
idx %>% tk_make_future_timeseries(length_out = 3)

# Make next 6 seconds of timestamps from the next timestamp
idx %>% tk_make_future_timeseries(length_out = "6 sec")


# Basic Example - By 1 Month
idx <- tk_make_timeseries("2016-01-01", by = "1 month",
                          length_out = "12 months")
idx

# Make 12 months of timestamps from the next timestamp
idx %>% tk_make_future_timeseries(length_out = "12 months")



# --- APPLICATION ---
# - Combine holiday sequences with future sequences

# Create index of days that FB stock will be traded in 2017 based on 2016 + holidays
FB_tbl <- FANG %>% dplyr::filter(symbol == "FB")
```

```
holidays <- tk_make_holiday_sequence(
    start_date = "2017-01-01",
    end_date   = "2017-12-31",
    calendar   = "NYSE")

# Remove holidays with skip_values, and remove weekends with inspect_weekdays = TRUE
FB_tbl %>%
    tk_index() %>%
    tk_make_future_timeseries(length_out       = "1 year",
                              inspect_weekdays = TRUE,
                              skip_values      = holidays)
```

---

tk_make_holiday_sequence

Make daily Holiday and Weekend date sequences

Description

Make daily Holiday and Weekend date sequences

Usage

```
tk_make_holiday_sequence(
  start_date,
  end_date,
  calendar = c("NYSE", "LONDON", "NERC", "TSX", "ZURICH"),
  skip_values = NULL,
  insert_values = NULL
)

tk_make_weekend_sequence(start_date, end_date)

tk_make_weekday_sequence(
  start_date,
  end_date,
  remove_weekends = TRUE,
  remove_holidays = FALSE,
  calendar = c("NYSE", "LONDON", "NERC", "TSX", "ZURICH"),
  skip_values = NULL,
  insert_values = NULL
)
```

Arguments

- `start_date` Used to define the starting date for date sequence generation. Provide in "YYYY-MM-DD" format.
- `end_date` Used to define the ending date for date sequence generation. Provide in "YYYY-MM-DD" format.
- `calendar` The calendar to be used in Date Sequence calculations for Holidays from the `timeDate` package. Acceptable values are: `"NYSE"`, `"LONDON"`, `"NERC"`, `"TSX"`, `"ZURICH"`.
- `skip_values` A daily date sequence to skip
- `insert_values` A daily date sequence to insert
- `remove_weekends` A logical value indicating whether or not to remove weekends (Saturday and Sunday) from the date sequence
- `remove_holidays` A logical value indicating whether or not to remove common holidays from the date sequence

Details

Start and End Date Specification

- Accept shorthand notation (i.e. `tk_make_timeseries()` specifications apply)
- Only available in Daily Periods.

Holiday Sequences

`tk_make_holiday_sequence()` is a wrapper for various holiday calendars from the `timeDate` package, making it easy to generate holiday sequences for common business calendars:

- New York Stock Exchange: `calendar = "NYSE"`
- Londo Stock Exchange: `"LONDON"`
- North American Reliability Council: `"NERC"`
- Toronto Stock Exchange: `"TSX"`
- Zurich Stock Exchange: `"ZURICH"`

Weekend and Weekday Sequences

These simply populate

Value

A vector containing future dates

See Also

- Intelligent date or date-time sequence creation: `tk_make_timeseries()`
- Holidays and weekends: `tk_make_holiday_sequence()`, `tk_make_weekend_sequence()`, `tk_make_weekday_sequence()`
- Make future index from existing: `tk_make_future_timeseries()`

Examples

```
library(dplyr)

# Set max.print to 50
options_old <- options()$max.print
options(max.print = 50)


# ---- HOLIDAYS & WEEKENDS ----

# Business Holiday Sequence
tk_make_holiday_sequence("2017-01-01", "2017-12-31", calendar = "NYSE")

tk_make_holiday_sequence("2017", calendar = "NYSE") # Same thing as above (just shorter)

# Weekday Sequence
tk_make_weekday_sequence("2017", "2018", remove_holidays = TRUE)

# Weekday Sequence + Removing Business Holidays
tk_make_weekday_sequence("2017", "2018", remove_holidays = TRUE)


# ---- COMBINE HOLIDAYS WITH MAKE FUTURE TIMESERIES FROM EXISTING ----
# - A common machine learning application is creating a future time series data set
#   from an existing

# Create index of days that FB stock will be traded in 2017 based on 2016 + holidays
FB_tbl <- FANG %>% dplyr::filter(symbol == "FB")

holidays <- tk_make_holiday_sequence(
    start_date = "2016",
    end_date   = "2017",
    calendar   = "NYSE")

weekends <- tk_make_weekend_sequence(
    start_date = "2016",
    end_date   = "2017")

# Remove holidays and weekends with skip_values
# We could also remove weekends with inspect_weekdays = TRUE
FB_tbl %>%
    tk_index() %>%
    tk_make_future_timeseries(length_out       = 366,
                              skip_values      = c(holidays, weekends))


options(max.print = options_old)
```

`tk_make_timeseries` *Intelligent date and date-time sequence creation*

Description

Improves on the seq.Date() and seq.POSIXt() functions by simplifying into 1 function tk_make_timeseries(). Intelligently handles character dates and logical assumptions based on user inputs.

Usage

```
tk_make_timeseries(
  start_date,
  end_date,
  by,
  length_out = NULL,
  include_endpoints = TRUE,
  skip_values = NULL,
  insert_values = NULL
)
```

Arguments

start_date	Used to define the starting date for date sequence generation. Provide in "YYYY-MM-DD" format.
end_date	Used to define the ending date for date sequence generation. Provide in "YYYY-MM-DD" format.
by	A character string, containing one of "sec", "min", "hour", "day", "week", "month", "quarter" or "year". You can create regularly spaced sequences using phrases like by = "10 min". See Details.
length_out	Optional length of the sequence. Can be used instead of one of: start_date, end_date, or by. Can be specified as a number or a time-based phrase.
include_endpoints	Logical. Whether or not to keep the last value when length_out is a time-based phrase. Default is TRUE (keep last value).
skip_values	A sequence to skip
insert_values	A sequence to insert

Details

The tk_make_timeseries() function handles both date and date-time sequences automatically.

- Parses date and date times from character
- Intelligently guesses the sequence desired based on arguments provided
- Handles spacing intelligently
- When both by and length_out are missing, guesses either second or day sequences
- Can skip and insert values if needed.

Start and End Date Specification

Start and end dates can be specified in reduced time-based phrases:

- start_date = "2014": Is converted to "2014-01-01" (start of period)
- end_date = "2014": Is converted to "2014-12-31" (end of period)
- start_date = "2014-03": Is converted to "2014-03-01" (start of period)
- end_date = "2014-03": Is converted to "2014-03-31" (end of period)

A similar process can be used for date-times.

By: Daily Sequences

Make a daily sequence with tk_make_timeseries(by). Examples:

- Every Day: by = "day"
- Every 2-Weeks: by = "2 weeks"
- Every 6-months: by = "6 months"

If missing, will guess by = "day"

By: Sub-Daily Sequences

Make a sub-daily sequence with tk_make_timeseries(by). Examples:

- Every minute: by = "min"
- Every 30-seconds: by = "30 sec"
- Every 2-hours: by = "2 hours

If missing, will guess by = "sec" if the start or end date is a date-time specification.

Length Out

The length_out can be specified by number of observation or complex time-based expressions. The following examples are all possible.

- length_out = 12 Creates 12 evenly spaced observations.
- length_out = "12 months" Adjusts the end date so it falls on the 12th month.

Include Endpoint

Sometimes the last date is not desired. For example, if the user specifies length_out = 12 months, the user may want the last value to be the 12th month and not the 13th. Just toggle, include_endpoint = FALSE to obtain this behavior.

Skip / Insert Timestamps

Skips and inserts are performed after the sequence is generated. This means that if you use the length_out parameter, the length may differ than the length_out.

Value

A vector containing date or date-times

See Also

- Intelligent date or date-time sequence creation: tk_make_timeseries()
- Holidays and weekends: tk_make_holiday_sequence(), tk_make_weekend_sequence(), tk_make_weekday_sequence()
- Make future index from existing: tk_make_future_timeseries()

Examples

```
library(dplyr)

# Set max.print to 50
options_old <- options()$max.print
options(max.print = 50)

# ---- DATE ----

# Start + End, Guesses by = "day"
tk_make_timeseries("2017-01-01", "2017-12-31")

# Just Start
tk_make_timeseries("2017") # Same result

# Only dates in February, 2017
tk_make_timeseries("2017-02")

# Start + Length Out, Guesses by = "day"
tk_make_timeseries("2012", length_out = 6) # Guesses by = "day"

# Start + By + Length Out, Spacing 6 observations by monthly interval
tk_make_timeseries("2012", by = "1 month", length_out = 6)

# Start + By + Length Out, Phrase "1 year 6 months"
tk_make_timeseries("2012", by = "1 month",
                   length_out = "1 year 6 months", include_endpoints = FALSE)

# Going in Reverse, End + Length Out
tk_make_timeseries(end_date = "2012-01-01", by = "1 month",
                   length_out = "1 year 6 months", include_endpoints = FALSE)

# ---- DATE-TIME ----

# Start + End, Guesses by second
tk_make_timeseries("2016-01-01 01:01:02", "2016-01-01 01:01:04")

# Date-Time Sequence - By 10 Minutes
# - Converts to date-time automatically & applies 10-min interval
tk_make_timeseries("2017-01-01", "2017-01-02", by = "10 min")


# --- REMOVE / INCLUDE ENDPOINTS ----

# Last value in this case is desired
tk_make_timeseries("2017-01-01", by = "30 min", length_out = "6 hours")

# Last value in monthly case is not wanted
tk_make_timeseries("2012-01-01", by = "1 month",
                   length_out = "12 months",
                   include_endpoints = FALSE) # Removes unnecessary last value
```

```
# ---- SKIP & INSERT VALUES ----

tk_make_timeseries(
    "2011-01-01", length_out = 5,
    skip_values   = "2011-01-05",
    insert_values = "2011-01-06"
)

options(max.print = options_old)
```

---

tk_seasonal_diagnostics

Group-wise Seasonality Data Preparation

Description

`tk_seasonal_diagnostics()` is the preprocessor for `plot_seasonal_diagnostics()`. It helps by automating feature collection for time series seasonality analysis.

Usage

```
tk_seasonal_diagnostics(.data, .date_var, .value, .feature_set = "auto")
```

Arguments

`.data` A `tibble` or `data.frame` with a time-based column

`.date_var` A column containing either date or date-time values

`.value` A column containing numeric values

`.feature_set` One or multiple selections to analyze for seasonality. Choices include:

- "auto" - Automatically selects features based on the time stamps and length of the series.
- "second" - Good for analyzing seasonality by second of each minute.
- "minute" - Good for analyzing seasonality by minute of the hour
- "hour" - Good for analyzing seasonality by hour of the day
- "wday.lbl" - Labeled weekdays. Good for analyzing seasonality by day of the week.
- "week" - Good for analyzing seasonality by week of the year.
- "month.lbl" - Labeled months. Good for analyzing seasonality by month of the year.
- "quarter" - Good for analyzing seasonality by quarter of the year
- "year" - Good for analyzing seasonality over multiple years.

Details

Automatic Feature Selection

Internal calculations are performed to detect a sub-range of features to include useing the following logic:

- The *minimum* feature is selected based on the median difference between consecutive timestamps
- The *maximum* feature is selected based on having 2 full periods.

Example: Hourly timestamp data that lasts more than 2 weeks will have the following features: "hour", "wday.lbl", and "week".

Scalable with Grouped Data Frames

This function respects grouped `data.frame` and `tibbles` that were made with `dplyr::group_by()`.

For grouped data, the automatic feature selection returned is a collection of all features within the sub-groups. This means extra features are returned even though they may be meaningless for some of the groups.

Transformations

The `.value` parameter respects transformations (e.g. `.value = log(sales)`).

Value

A `tibble` or `data.frame` with seasonal features

Examples

```
library(dplyr)

# ---- GROUPED EXAMPLES ----

# Hourly Data
m4_hourly %>%
    group_by(id) %>%
    tk_seasonal_diagnostics(date, value)

# Monthly Data
m4_monthly %>%
    group_by(id) %>%
    tk_seasonal_diagnostics(date, value)

# ---- TRANSFORMATION ----

m4_weekly %>%
    group_by(id) %>%
    tk_seasonal_diagnostics(date, log(value))

# ---- CUSTOM FEATURE SELECTION ----

m4_hourly %>%
    group_by(id) %>%
```

```
tk_seasonal_diagnostics(date, value, .feature_set = c("hour", "week"))
```

`tk_stl_diagnostics` *Group-wise STL Decomposition (Season, Trend, Remainder)*

Description

`tk_stl_diagnostics()` is the preprocessor for `plot_stl_diagnostics()`. It helps by automating frequency and trend selection.

Usage

```
tk_stl_diagnostics(
  .data,
  .date_var,
  .value,
  .frequency = "auto",
  .trend = "auto",
  .message = TRUE
)
```

Arguments

`.data`	A `tibble` or `data.frame` with a time-based column
`.date_var`	A column containing either date or date-time values
`.value`	A column containing numeric values
`.frequency`	Controls the seasonal adjustment (removal of seasonality). Input can be either "auto", a time-based definition (e.g. "2 weeks"), or a numeric number of observations per frequency (e.g. 10). Refer to `tk_get_frequency()`.
`.trend`	Controls the trend component. For STL, trend controls the sensitivity of the lowess smoother, which is used to remove the remainder.
`.message`	A boolean. If `TRUE`, will output information related to automatic frequency and trend selection (if applicable).

Details

The `tk_stl_diagnostics()` function generates a Seasonal-Trend-Loess decomposition. The function is "tidy" in the sense that it works on data frames and is designed to work with `dplyr` groups.

STL method:

The STL method implements time series decomposition using the underlying `stats::stl()`. The decomposition separates the "season" and "trend" components from the "observed" values leaving the "remainder".

Frequency & Trend Selection

The user can control two parameters: `.frequency` and `.trend`.

1. The `.frequency` parameter adjusts the "season" component that is removed from the "observed" values.
2. The `.trend` parameter adjusts the trend window (`t.window` parameter from `stl()`) that is used.

The user may supply both `.frequency` and `.trend` as time-based durations (e.g. "6 weeks") or numeric values (e.g. 180) or "auto", which automatically selects the frequency and/or trend based on the scale of the time series.

Value

A `tibble` or `data.frame` with Observed, Season, Trend, Remainder, and Seasonally-Adjusted features

Examples

```
library(dplyr)


# ---- GROUPS & TRANSFORMATION ----
m4_daily %>%
    group_by(id) %>%
    tk_stl_diagnostics(date, box_cox_vec(value))

# ---- CUSTOM TREND ----
m4_weekly %>%
    group_by(id) %>%
    tk_stl_diagnostics(date, box_cox_vec(value), .trend = "2 quarters")
```

`tk_summary_diagnostics` *Group-wise Time Series Summary*

Description

`tk_summary_diagnostics()` returns the time series summary from one or more timeseries groups in a tibble.

Usage

```
tk_summary_diagnostics(.data, .date_var)
```

Arguments

`.data`	A `tibble` or `data.frame` with a time-based column
`.date_var`	A column containing either date or date-time values. If missing, attempts to auto-detect the date or date-time column.

Details

Applies `tk_get_timeseries_summary()` group-wise returning the summary of one or more time series groups.

- Respects `dplyr` groups
- Returns the time series summary from a time-based feature.

Value

A `tibble` or `data.frame` with timeseries summary features

Examples

```
library(dplyr)

# ---- NON-GROUPED EXAMPLES ----

# Monthly Data
m4_monthly %>%
    filter(id == "M750") %>%
    tk_summary_diagnostics()

# ---- GROUPED EXAMPLES ----

# Monthly Data
m4_monthly %>%
    group_by(id) %>%
    tk_summary_diagnostics()
```

---

tk_tbl — *Coerce time-series objects to tibble.*

Description

Coerce time-series objects to tibble.

Usage

```
tk_tbl(
  data,
  preserve_index = TRUE,
  rename_index = "index",
  timetk_idx = FALSE,
  silent = FALSE,
  ...
)
```

Arguments

data	A time-series object.
preserve_index	Attempts to preserve a time series index. Default is TRUE.
rename_index	Enables the index column to be renamed.
timetk_idx	Used to return a date / datetime index for regularized objects that contain a timetk "index" attribute. Refer to tk_index() for more information on returning index information from regularized timeseries objects (i.e. ts).
silent	Used to toggle printing of messages and warnings.
...	Additional parameters passed to the tibble::as_tibble() function.

Details

tk_tbl is designed to coerce time series objects (e.g. xts, zoo, ts, timeSeries, etc) to tibble objects. The main advantage is that the function keeps the date / date-time information from the underlying time-series object.

When preserve_index = TRUE is specified, a new column, index, is created during object coercion, and the function attempts to preserve the date or date-time information. The date / date-time column name can be changed using the rename_index argument.

The timetk_idx argument is applicable when coercing ts objects that were created using tk_ts() from an object that had a time base (e.g. tbl, xts, zoo). Setting timetk_idx = TRUE enables returning the timetk "index" attribute if present, which is the original (non-regularized) time-based index.

Value

Returns a tibble object.

See Also

tk_xts(), tk_zoo(), tk_zooreg(), tk_ts()

Examples

```
library(dplyr)

data_tbl <- tibble(
    date = seq.Date(from = as.Date("2010-01-01"), by = 1, length.out = 5),
    x    = seq(100, 120, by = 5)
)


### ts to tibble: Comparison between as.data.frame() and tk_tbl()
data_ts <- tk_ts(data_tbl, start = c(2010,1), freq = 365)

# No index
as.data.frame(data_ts)

# Defualt index returned is regularized numeric index
```

```
tk_tbl(data_ts)

# Original date index returned (Only possible if original data has time-based index)
tk_tbl(data_ts, timetk_idx = TRUE)


### xts to tibble: Comparison between as.data.frame() and tk_tbl()
data_xts <- tk_xts(data_tbl)

# Dates are character class stored in row names
as.data.frame(data_xts)

# Dates are appropriate date class and within the data frame
tk_tbl(data_xts)


### zooreg to tibble: Comparison between as.data.frame() and tk_tbl()
data_zooreg <- tk_zooreg(1:8, start = zoo::yearqtr(2000), frequency = 4)

# Dates are character class stored in row names
as.data.frame(data_zooreg)

# Dates are appropriate zoo yearqtr class within the data frame
tk_tbl(data_zooreg)


### zoo to tibble: Comparison between as.data.frame() and tk_tbl()
data_zoo <- zoo::zoo(1:12, zoo::yearmon(2016 + seq(0, 11)/12))

# Dates are character class stored in row names
as.data.frame(data_zoo)

# Dates are appropriate zoo yearmon class within the data frame
tk_tbl(data_zoo)
```

---

tk_time_series_cv_plan

Time Series Resample Plan Data Preparation

Description

The `tk_time_series_cv_plan()` function provides a simple interface to prepare a time series resample specification (`rset`) of either `rolling_origin` or `time_series_cv` class.

Usage

```
tk_time_series_cv_plan(.data)
```

Arguments

`.data`	A time series resample specification of of either `rolling_origin` or `time_series_cv` class.

Details

Resample Set

A resample set is an output of the `timetk::time_series_cv()` function or the `rsample::rolling_origin()` function.

Value

A `tibble` containing the time series crossvalidation plan.

See Also

- `time_series_cv()` and `rsample::rolling_origin()` - Functions used to create time series resample specfications.
- `plot_time_series_cv_plan()` - The plotting function used for visualizing the time series resample plan.

Examples

```
library(dplyr)
library(rsample)

FB_tbl <- FANG %>%
    filter(symbol == "FB") %>%
    select(symbol, date, adjusted)

resample_spec <- time_series_cv(
    FB_tbl,
    initial = 150, assess = 50, skip = 50,
    cumulative = FALSE,
    lag = 30,
    slice_limit = n())

resample_spec %>% tk_time_series_cv_plan()
```

---

tk_ts — *Coerce time series objects and tibbles with date/date-time columns to ts.*

Description

Coerce time series objects and tibbles with date/date-time columns to ts.

Usage

```
tk_ts(
  data,
  select = NULL,
  start = 1,
  end = numeric(),
  frequency = 1,
  deltat = 1,
  ts.eps = getOption("ts.eps"),
  silent = FALSE
)

tk_ts_(
  data,
  select = NULL,
  start = 1,
  end = numeric(),
  frequency = 1,
  deltat = 1,
  ts.eps = getOption("ts.eps"),
  silent = FALSE
)
```

Arguments

`data`	A time-based tibble or time-series object.
`select`	**Applicable to tibbles and data frames only**. The column or set of columns to be coerced to `ts` class.
`start`	the time of the first observation. Either a single number or a vector of two numbers (the second of which is an integer), which specify a natural time unit and a (1-based) number of samples into the time unit. See the examples for the use of the second form.
`end`	the time of the last observation, specified in the same way as `start`.
`frequency`	the number of observations per unit of time.
`deltat`	the fraction of the sampling period between successive observations; e.g., 1/12 for monthly data. Only one of `frequency` or `deltat` should be provided.
`ts.eps`	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than `ts.eps`.
`silent`	Used to toggle printing of messages and warnings.

Details

`tk_ts()` is a wrapper for `stats::ts()` that is designed to coerce `tibble` objects that have a "time-base" (meaning the values vary with time) to `ts` class objects. There are two main advantages:

1. Non-numeric columns get removed instead of being populated by NA's.

2. The returned `ts` object retains a "timetk index" (and various other attributes) if detected. The "timetk index" can be used to coerce between `tbl`, `xts`, `zoo`, and `ts` data types.

The `select` argument is used to select subsets of columns from the incoming data.frame. Only columns containing numeric data are coerced. *At a minimum, a* `frequency` *and a* `start` *should be specified.*

For non-data.frame object classes (e.g. `xts`, `zoo`, `timeSeries`, etc) the objects are coerced using `stats::ts()`.

`tk_ts_` is a nonstandard evaluation method.

Value

Returns a `ts` object.

See Also

`tk_index()`, `tk_tbl()`, `tk_xts()`, `tk_zoo()`, `tk_zooreg()`

Examples

```
library(dplyr)

### tibble to ts: Comparison between tk_ts() and stats::ts()
data_tbl <- tibble::tibble(
    date = seq.Date(as.Date("2016-01-01"), by = 1, length.out = 5),
    x    = rep("chr values", 5),
    y    = cumsum(1:5),
    z    = cumsum(11:15) * rnorm(1))

# as.ts: Character columns introduce NA's; Result does not retain index
stats::ts(data_tbl[,-1], start = 2016)

# tk_ts: Only numeric columns get coerced; Result retains index in numeric format
data_ts <- tk_ts(data_tbl, start = 2016)
data_ts

# timetk index
tk_index(data_ts, timetk_idx = FALSE)   # Regularized index returned
tk_index(data_ts, timetk_idx = TRUE)    # Original date index returned

# Coerce back to tibble
data_ts %>% tk_tbl(timetk_idx = TRUE)


### Using select
tk_ts(data_tbl, select = y)


### NSE: Enables programming
select   <- "y"
tk_ts_(data_tbl, select = select)
```

tk_tsfeatures *Time series feature matrix (Tidy)*

Description

`tk_tsfeatures()` is a tidyverse compliant wrapper for `tsfeatures::tsfeatures()`. The function computes a matrix of time series features that describes the various time series. It's designed for groupwise analysis using `dplyr` groups.

Usage

```
tk_tsfeatures(
  .data,
  .date_var,
  .value,
  .period = "auto",
  .features = c("frequency", "stl_features", "entropy", "acf_features"),
  .scale = TRUE,
  .trim = FALSE,
  .trim_amount = 0.1,
  .parallel = FALSE,
  .na_action = na.pass,
  .prefix = "ts_",
  .silent = TRUE,
  ...
)
```

Arguments

`.data` A `tibble` or `data.frame` with a time-based column

`.date_var` A column containing either date or date-time values

`.value` A column containing numeric values

`.period` The periodicity (frequency) of the time series data. Values can be provided as follows:

- "auto" (default) Calculates using `tk_get_frequency()`.
- "2 weeks": Would calculate the median number of observations in a 2-week window.
- 7 (numeric): Would interpret the `ts` frequency as 7 observations per cycle (common for weekly data)

`.features` Passed to `features` in the underlying `tsfeatures()` function. A vector of function names that represent a feature aggregation function. Examples:

1. Use one of the function names from `tsfeatures` R package e.g.("lumpiness", "stl_features").
2. Use a function name (e.g. "mean" or "median")
3. Create your own function and provide the function name

`.scale`	If `TRUE`, time series are scaled to mean 0 and sd 1 before features are computed.
`.trim`	If `TRUE`, time series are trimmed by trim_amount before features are computed. Values larger than trim_amount in absolute value are set to `NA`.
`.trim_amount`	Default level of trimming if trim==TRUE. Default: 0.1.
`.parallel`	If TRUE, multiple cores (or multiple sessions) will be used. This only speeds things up when there are a large number of time series. When `.parallel = TRUE`, the `multiprocess = future::multisession`. This can be adjusted by setting `multiprocess` parameter. See the `tsfeatures::tsfeatures()` function for mor details.
`.na_action`	A function to handle missing values. Use na.interp to estimate missing values.
`.prefix`	A prefix to prefix the feature columns. Default: `"ts_"`.
`.silent`	Whether or not to show messages and warnings.
`...`	Other arguments get passed to the feature functions.

Details

The `timetk::tk_tsfeatures()` function implements the `tsfeatures` package for computing aggregated feature matrix for time series that is useful in many types of analysis such as clustering time series.

The `timetk` version ports the `tsfeatures::tsfeatures()` function to a `tidyverse`-compliant format that uses a tidy data frame containing grouping columns (optional), a date column, and a value column. Other columns are ignored.

It then becomes easy to summarize each time series by group-wise application of `.features`, which are simply functions that evaluate a time series and return single aggregated value. (Example: "mean" would return the mean of the time series (note that values are scaled to mean 1 and sd 0 first))

Function Internals:

Internally, the time series are converted to `ts` class using `tk_ts(.period)` where the period is the frequency of the time series. Values can be provided for `.period`, which will be used prior to convertion to `ts` class.

The function then leverages `tsfeatures::tsfeatures()` to compute the feature matrix of summarized feature values.

Value

A `tibble` or `data.frame` with aggregated features that describe each time series.

References

1. Rob Hyndman, Yanfei Kang, Pablo Montero-Manso, Thiyanga Talagala, Earo Wang, Yangzhuoran Yang, Mitchell O'Hara-Wild: tsfeatures R package

Examples

```
library(dplyr)

walmart_sales_weekly %>%
    group_by(id) %>%
    tk_tsfeatures(
      .date_var = Date,
      .value    = Weekly_Sales,
      .period   = 52,
      .features = c("frequency", "stl_features", "entropy", "acf_features", "mean"),
      .scale    = TRUE,
      .prefix   = "ts_"
    )
```

`tk_xts` *Coerce time series objects and tibbles with date/date-time columns to xts.*

Description

Coerce time series objects and tibbles with date/date-time columns to xts.

Usage

```
tk_xts(data, select = NULL, date_var = NULL, silent = FALSE, ...)

tk_xts_(data, select = NULL, date_var = NULL, silent = FALSE, ...)
```

Arguments

`data`	A time-based tibble or time-series object.
`select`	**Applicable to tibbles and data frames only**. The column or set of columns to be coerced to `ts` class.
`date_var`	**Applicable to tibbles and data frames only**. Column name to be used to `order.by`. `NULL` by default. If `NULL`, function will find the date or date-time column.
`silent`	Used to toggle printing of messages and warnings.
`...`	Additional parameters to be passed to `xts::xts()`. Refer to `xts::xts()`.

Details

`tk_xts` is a wrapper for `xts::xts()` that is designed to coerce `tibble` objects that have a "time-base" (meaning the values vary with time) to `xts` class objects. There are three main advantages:

1. Non-numeric columns that are not removed via `select` are dropped and the user is warned. This prevents an error or coercion issue from occurring.

2. The date column is auto-detected if not specified by date_var. This takes the effort off the user to assign a date vector during coercion.
3. ts objects are automatically coerced if a "timetk index" is present. Refer to tk_ts().

The select argument can be used to select subsets of columns from the incoming data.frame. Only columns containing numeric data are coerced. The date_var can be used to specify the column with the date index. If date_var = NULL, the date / date-time column is interpreted. Optionally, the order.by argument from the underlying xts::xts() function can be used. The user must pass a vector of dates or date-times if order.by is used.

For non-data.frame object classes (e.g. xts, zoo, timeSeries, etc) the objects are coerced using xts::xts().

tk_xts_ is a nonstandard evaluation method.

Value

Returns a xts object.

See Also

tk_tbl(), tk_zoo(), tk_zooreg(), tk_ts()

Examples

```
library(dplyr)

### tibble to xts: Comparison between tk_xts() and xts::xts()
data_tbl <- tibble::tibble(
    date = seq.Date(as.Date("2016-01-01"), by = 1, length.out = 5),
    x    = rep("chr values", 5),
    y    = cumsum(1:5),
    z    = cumsum(11:15) * rnorm(1))

# xts: Character columns cause coercion issues; order.by must be passed a vector of dates
xts::xts(data_tbl[,-1], order.by = data_tbl$date)

# tk_xts: Non-numeric columns automatically dropped; No need to specify date column
tk_xts(data_tbl)

# ts can be coerced back to xts
data_tbl %>%
    tk_ts(start = 2016, freq = 365) %>%
    tk_xts()

### Using select and date_var
tk_xts(data_tbl, select = y, date_var = date)


### NSE: Enables programming
date_var <- "date"
select   <- "y"
tk_xts_(data_tbl, select = select, date_var = date_var)
```

tk_zoo	*Coerce time series objects and tibbles with date/date-time columns to xts.*

Description

Coerce time series objects and tibbles with date/date-time columns to xts.

Usage

```
tk_zoo(data, select = NULL, date_var = NULL, silent = FALSE, ...)

tk_zoo_(data, select = NULL, date_var = NULL, silent = FALSE, ...)
```

Arguments

`data`	A time-based tibble or time-series object.
`select`	**Applicable to tibbles and data frames only**. The column or set of columns to be coerced to `ts` class.
`date_var`	**Applicable to tibbles and data frames only**. Column name to be used to `order.by`. `NULL` by default. If `NULL`, function will find the date or date-time column.
`silent`	Used to toggle printing of messages and warnings.
`...`	Additional parameters to be passed to `xts::xts()`. Refer to `xts::xts()`.

Details

`tk_zoo` is a wrapper for `zoo::zoo()` that is designed to coerce `tibble` objects that have a "time-base" (meaning the values vary with time) to `zoo` class objects. There are three main advantages:

1. Non-numeric columns that are not removed via `select` are dropped and the user is warned. This prevents an error or coercion issue from occurring.
2. The date column is auto-detected if not specified by `date_var`. This takes the effort off the user to assign a date vector during coercion.
3. `ts` objects are automatically coerced if a "timetk index" is present. Refer to `tk_ts()`.

The `select` argument can be used to select subsets of columns from the incoming data.frame. Only columns containing numeric data are coerced. The `date_var` can be used to specify the column with the date index. If `date_var = NULL`, the date / date-time column is interpreted. Optionally, the `order.by` argument from the underlying `zoo::zoo()` function can be used. The user must pass a vector of dates or date-times if `order.by` is used. *Important Note: The* `...` *arguments are passed to* `xts::xts()`*, which enables additional information (e.g. time zone) to be an attribute of the* `zoo` *object.*

For non-data.frame object classes (e.g. `xts`, `zoo`, `timeSeries`, etc) the objects are coerced using `zoo::zoo()`.

`tk_zoo_` is a nonstandard evaluation method.

Value

Returns a zoo object.

See Also

tk_tbl(), tk_xts(), tk_zooreg(), tk_ts()

Examples

```
library(dplyr)

### tibble to zoo: Comparison between tk_zoo() and zoo::zoo()
data_tbl <- dplyr::tibble(
    date = seq.Date(as.Date("2016-01-01"), by = 1, length.out = 5),
    x    = rep("chr values", 5),
    y    = cumsum(1:5),
    z    = cumsum(11:15) * rnorm(1))

# zoo: Characters will cause error; order.by must be passed a vector of dates
zoo::zoo(data_tbl[,-c(1,2)], order.by = data_tbl$date)

# tk_zoo: Character columns dropped with a warning; No need to specify dates (auto detected)
tk_zoo(data_tbl)

# ts can be coerced back to zoo
data_tbl %>%
    tk_ts(start = 2016, freq = 365) %>%
    tk_zoo()


### Using select and date_var
tk_zoo(data_tbl, select = y, date_var = date)


### NSE: Enables programming
date_var <- "date"
select   <- "y"
tk_zoo_(data_tbl, select = select, date_var = date_var)
```

---

tk_zooreg *Coerce time series objects and tibbles with date/date-time columns to ts.*

Description

Coerce time series objects and tibbles with date/date-time columns to ts.

Usage

```
tk_zooreg(
  data,
  select = NULL,
  date_var = NULL,
  start = 1,
  end = numeric(),
  frequency = 1,
  deltat = 1,
  ts.eps = getOption("ts.eps"),
  order.by = NULL,
  silent = FALSE
)

tk_zooreg_(
  data,
  select = NULL,
  date_var = NULL,
  start = 1,
  end = numeric(),
  frequency = 1,
  deltat = 1,
  ts.eps = getOption("ts.eps"),
  order.by = NULL,
  silent = FALSE
)
```

Arguments

`data`	A time-based tibble or time-series object.
`select`	**Applicable to tibbles and data frames only**. The column or set of columns to be coerced to `zooreg` class.
`date_var`	**Applicable to tibbles and data frames only**. Column name to be used to `order.by`. `NULL` by default. If `NULL`, function will find the date or date-time column.
`start`	the time of the first observation. Either a single number or a vector of two integers, which specify a natural time unit and a (1-based) number of samples into the time unit.
`end`	the time of the last observation, specified in the same way as `start`.
`frequency`	the number of observations per unit of time.
`deltat`	the fraction of the sampling period between successive observations; e.g., 1/12 for monthly data. Only one of `frequency` or `deltat` should be provided.
`ts.eps`	time series comparison tolerance. Frequencies are considered equal if their absolute difference is less than `ts.eps`.
`order.by`	a vector by which the observations in `x` are ordered. If this is specified the arguments `start` and `end` are ignored and `zoo(data, order.by, frequency)` is called. See `zoo` for more information.

silent Used to toggle printing of messages and warnings.

Details

tk_zooreg() is a wrapper for zoo::zooreg() that is designed to coerce tibble objects that have a "time-base" (meaning the values vary with time) to zooreg class objects. There are two main advantages:

1. Non-numeric columns get removed instead causing coercion issues.
2. If an index is present, the returned zooreg object retains an index retrievable using tk_index().

The select argument is used to select subsets of columns from the incoming data.frame. The date_var can be used to specify the column with the date index. If date_var = NULL, the date / date-time column is interpreted. Optionally, the order.by argument from the underlying xts::xts() function can be used. The user must pass a vector of dates or date-times if order.by is used. Only columns containing numeric data are coerced. *At a minimum, a* frequency *and a* start *should be specified.*

For non-data.frame object classes (e.g. xts, zoo, timeSeries, etc) the objects are coerced using zoo::zooreg().

tk_zooreg_ is a nonstandard evaluation method.

Value

Returns a zooreg object.

See Also

tk_tbl(), tk_xts(), tk_zoo(), tk_ts()

Examples

```
### tibble to zooreg: Comparison between tk_zooreg() and zoo::zooreg()
data_tbl <- tibble::tibble(
    date = seq.Date(as.Date("2016-01-01"), by = 1, length.out = 5),
    x    = rep("chr values", 5),
    y    = cumsum(1:5),
    z    = cumsum(11:15) * rnorm(1))

# zoo::zooreg: Values coerced to character; Result does not retain index
data_zooreg <- zoo::zooreg(data_tbl[,-1], start = 2016, freq = 365)
data_zooreg                # Numeric values coerced to character
rownames(data_zooreg)      # NULL, no dates retained

# tk_zooreg: Only numeric columns get coerced; Result retains index as rownames
data_tk_zooreg <- tk_zooreg(data_tbl, start = 2016, freq = 365)
data_tk_zooreg             # No inadvertent coercion to character class

# timetk index
tk_index(data_tk_zooreg, timetk_idx = FALSE)   # Regularized index returned
tk_index(data_tk_zooreg, timetk_idx = TRUE)    # Original date index returned
```

```
### Using select and date_var
tk_zooreg(data_tbl, select = y, date_var = date, start = 2016, freq = 365)


### NSE: Enables programming
select   <- "y"
date_var <- "date"
tk_zooreg_(data_tbl, select = select, date_var = date_var, start = 2016, freq = 365)
```

`ts_clean_vec` *Replace Outliers & Missing Values in a Time Series*

Description

This is mainly a wrapper for the outlier cleaning function, `tsclean()`, from the `forecast` R package. The `ts_clean_vec()` function includes arguments for applying seasonality to numeric vector (non-`ts`) via the `period` argument.

Usage

```
ts_clean_vec(x, period = 1, lambda = NULL)
```

Arguments

`x`	A numeric vector.
`period`	A seasonal period to use during the transformation. If `period = 1`, seasonality is not included and `supsmu()` is used to fit a trend. If `period > 1`, a robust STL decomposition is first performed and a linear interpolation is applied to the seasonally adjusted data.
`lambda`	A box cox transformation parameter. If set to `"auto"`, performs automated lambda selection.

Details

Cleaning Outliers

1. Non-Seasonal (`period = 1`): Uses `stats::supsmu()`
2. Seasonal (`period > 1`): Uses `forecast::mstl()` with `robust = TRUE` (robust STL decomposition) for seasonal series.

To estimate missing values and outlier replacements, linear interpolation is used on the (possibly seasonally adjusted) series. See `forecast::tsoutliers()` for the outlier detection method.

Box Cox Transformation

In many circumstances, a Box Cox transformation can help. Especially if the series is multiplicative meaning the variance grows exponentially. A Box Cox transformation can be automated by setting `lambda = "auto"` or can be specified by setting `lambda = numeric value`.

Value

A `numeric` vector with the missing values and/or anomalies transformed to imputed values.

References

- Forecast R Package
- Forecasting Principles & Practices: Dealing with missing values and outliers

See Also

- Box Cox Transformation: `box_cox_vec()`
- Lag Transformation: `lag_vec()`
- Differencing Transformation: `diff_vec()`
- Rolling Window Transformation: `slidify_vec()`
- Loess Smoothing Transformation: `smooth_vec()`
- Fourier Series: `fourier_vec()`
- Missing Value Imputation for Time Series: `ts_impute_vec()`
- Outlier Cleaning for Time Series: `ts_clean_vec()`

Examples

```
library(dplyr)


# --- VECTOR ----

values <- c(1,2,3, 4*2, 5,6,7, NA, 9,10,11, 12*2)
values

# Linear interpolation + Outlier Cleansing
ts_clean_vec(values, period = 1, lambda = NULL)

# Seasonal Interpolation: set period = 4
ts_clean_vec(values, period = 4, lambda = NULL)

# Seasonal Interpolation with Box Cox Transformation (internal)
ts_clean_vec(values, period = 4, lambda = "auto")
```

`ts_impute_vec` *Missing Value Imputation for Time Series*

Description

This is mainly a wrapper for the Seasonally Adjusted Missing Value using Linear Interpolation function, `na.interp()`, from the `forecast` R package. The `ts_impute_vec()` function includes arguments for applying seasonality to numeric vector (non-`ts`) via the `period` argument.

Usage

```
ts_impute_vec(x, period = 1, lambda = NULL)
```

Arguments

`x`	A numeric vector.
`period`	A seasonal period to use during the transformation. If `period = 1`, linear interpolation is performed. If `period > 1`, a robust STL decomposition is first performed and a linear interpolation is applied to the seasonally adjusted data.
`lambda`	A box cox transformation parameter. If set to `"auto"`, performs automated lambda selection.

Details

Imputation using Linear Interpolation

Three circumstances cause strictly linear interpolation:

1. **Period is 1:** With `period = 1`, a seasonality cannot be interpreted and therefore linear is used.
2. **Number of Non-Missing Values is less than 2-Periods**: Insufficient values exist to detect seasonality.
3. **Number of Total Values is less than 3-Periods**: Insufficient values exist to detect seasonality.

Seasonal Imputation using Linear Interpolation

For seasonal series with `period > 1`, a robust Seasonal Trend Loess (STL) decomposition is first computed. Then a linear interpolation is applied to the seasonally adjusted data, and the seasonal component is added back.

Box Cox Transformation

In many circumstances, a Box Cox transformation can help. Especially if the series is multiplicative meaning the variance grows exponentially. A Box Cox transformation can be automated by setting `lambda = "auto"` or can be specified by setting `lambda = numeric value`.

Value

A `numeric` vector with the missing values imputed.

References

- Forecast R Package
- Forecasting Principles & Practices: Dealing with missing values and outliers

See Also

- Box Cox Transformation: box_cox_vec()
- Lag Transformation: lag_vec()
- Differencing Transformation: diff_vec()
- Rolling Window Transformation: slidify_vec()
- Loess Smoothing Transformation: smooth_vec()
- Fourier Series: fourier_vec()
- Missing Value Imputation for Time Series: ts_impute_vec()

Examples

```
library(dplyr)


# --- VECTOR ----

values <- c(1,2,3, 4*2, 5,6,7, NA, 9,10,11, 12*2)
values

# Linear interpolation
ts_impute_vec(values, period = 1, lambda = NULL)

# Seasonal Interpolation: set period = 4
ts_impute_vec(values, period = 4, lambda = NULL)

# Seasonal Interpolation with Box Cox Transformation (internal)
ts_impute_vec(values, period = 4, lambda = "auto")
```

---

walmart_sales_weekly   *Sample Time Series Retail Data from the Walmart Recruiting Store Sales Forecasting Competition*

Description

The Kaggle "Walmart Recruiting - Store Sales Forecasting" Competition used **retail data** for combinations of stores and departments within each store. The competition began February 20th, 2014 and ended May 5th, 2014. The competition included data from 45 retail stores located in different regions. The dataset included various external features including Holiday information, Temperature, Fuel Price, and Markdown. This dataset includes a **Sample of 7 departments from the Store ID 1 (7 total time series)**.

Usage

```
walmart_sales_weekly
```

Format

A tibble: 9,743 x 3

- `id` Factor. Unique series identifier (4 total)
- `Store` Numeric. Store ID.
- `Dept` Numeric. Department ID.
- `Date` Date. Weekly timestamp.
- `Weekly_Sales` Numeric. Sales for the given department in the given store.
- `IsHoliday` Logical. Whether the week is a "special" holiday for the store.
- `Type` Character. Type identifier of the store.
- `Size` Numeric. Store square-footage
- `Temperature` Numeric. Average temperature in the region.
- `Fuel_Price` Numeric. Cost of fuel in the region.
- `MarkDown1`, `MarkDown2`, `MarkDown3`, `MarkDown4`, `MarkDown5` Numeric. Anonymized data related to promotional markdowns that Walmart is running. MarkDown data is only available after Nov 2011, and is not available for all stores all the time. Any missing value is marked with an NA.
- `CPI` Numeric. The consumer price index.
- `Unemployment` Numeric. The unemployment rate in the region.

Details

This is a sample of 7 Weekly data sets from the Kaggle Walmart Recruiting Store Sales Forecasting competition.

Holiday Features

The four holidays fall within the following weeks in the dataset (not all holidays are in the data):

- Super Bowl: 12-Feb-10, 11-Feb-11, 10-Feb-12, 8-Feb-13
- Labor Day: 10-Sep-10, 9-Sep-11, 7-Sep-12, 6-Sep-13
- Thanksgiving: 26-Nov-10, 25-Nov-11, 23-Nov-12, 29-Nov-13
- Christmas: 31-Dec-10, 30-Dec-11, 28-Dec-12, 27-Dec-13

Source

- Kaggle Competition Website

Examples

```
walmart_sales_weekly
```

wikipedia_traffic_daily

Sample Daily Time Series Data from the Web Traffic Forecasting (Wikipedia) Competition

Description

The Kaggle "Web Traffic Forecasting" (Wikipedia) Competition used **Google Analytics Web Traffic Data** for 145,000 websites. Each of these time series represent a number of daily views of a different Wikipedia articles. The competition began July 13th, 2017 and ended November 15th, 2017. This dataset includes a **Sample of 10 article pages (10 total time series)**.

Usage

```
wikipedia_traffic_daily
```

Format

A tibble: 9,743 x 3

- `Page` Character. Page information.
- `date` Date. Daily timestamp.
- `value` Numeric. Daily views of the wikipedia article.

Details

This is a sample of 10 Daily data sets from the Kaggle Web Traffic Forecasting (Wikipedia) Competition

Source

- Kaggle Competition Website

Examples

```
wikipedia_traffic_daily
```

Index