Package 'SporeLag'

August 4, 2026

Type Package

Title Lagged and Moving-Average Exposure Features for Aeroallergen Epidemiology

Version 0.1.1

Description Deterministic, group-safe utilities that transform daily environmental exposure series (pollen and spore counts, with support for other time-varying exposures such as ozone and particulate matter) into analysis-ready lagged and moving-average exposure features. Functions validate temporal regularity, assign ISO 8601 weeks and configurable seasons, impute missing daily values transparently, and construct lagged and windowed exposures suitable for environmental epidemiology and public health analyses.

URL https://github.com/friveramariani/SporeLag

BugReports https://github.com/friveramariani/SporeLag/issues

License MIT + file LICENSE

Encoding UTF-8

Suggests dplyr, knitr, lubridate, rmarkdown, spelling, testthat (>= 3.0.0)

Config/testthat/edition 3

RoxygenNote 8.0.0

Imports cli, rlang, stats

Depends R (>= 4.1.0)

LazyData true

VignetteBuilder knitr

Language en-US

NeedsCompilation no

Author Félix E. Rivera-Mariani [aut, cre] (ORCID: <https://orcid.org/0000-0002-6671-0174>)

Maintainer Félix E. Rivera-Mariani <felixrm@friveram.com>

Repository CRAN

Date/Publication 2026-08-04 09:30:02 UTC

Contents

apply_lag 2
assign_iso_week 3
assign_season 5
build_moving_average 7
complete_daily_grid 8
impute_weekly_mean 10
pollen_demo 12

Index **14**

`apply_lag` *Apply time-indexed lags to an exposure series*

Description

`apply_lag()` appends one column per requested lag, shifting the exposure series backwards in time within each group. A lag of `n` places the value observed `n` days earlier alongside the current day.

Usage

```
apply_lag(data, value, lags, date, by = character())
```

Arguments

`data`	A `data.frame` on a complete daily grid.
`value`	A single string naming a numeric exposure column.
`lags`	A vector of non-negative whole numbers, e.g. `0:3`.
`date`	A single string naming a `Date` column.
`by`	A character vector of grouping column names. Default `character()` (no grouping — the whole frame is one series).

Details

This function indexes by position, and therefore requires a complete daily grid. On a gapped series, "lag 1" would mean "the previous *row*", which may be one day earlier or thirty. The resulting exposure column looks entirely plausible, runs cleanly through a regression, and yields a silently misaligned lag–response estimate. `apply_lag()` therefore refuses to operate on a gapped grid and raises `sporelag_error_gaps`. Call `complete_daily_grid()` first. This is a deliberate design decision, not an oversight; see `vignette("getting-started")`.

Documented defaults:

- Lags are computed strictly **within** each group defined by `by`. Values never carry across a group boundary — the first `n` days of each site's series are `NA`, not the tail of the previous site's series.

- `lags = 0` is permitted and yields a copy of the same-day value. This exists so a uniform model matrix (`lag0, lag1, lag2, ...`) can be built without special-casing the contemporaneous term.
- **Negative values are an error.** A negative lag is a *lead*: it aligns an exposure with an outcome that preceded it. That is almost never intended, and when it is (e.g. a negative-control exposure), it should be written explicitly rather than arrived at by a sign slip.
- Row order is preserved. Columns are appended.

Value

`data` with one column appended per lag, named `{value}_lag{n}`. Row order and all input columns are unchanged.

Errors

`sporelag_error_gaps` The daily grid is incomplete within at least one group. Call `complete_daily_grid()` first.

`sporelag_error_duplicate_dates` A date appears more than once within a group.

`sporelag_error_bad_input` Bad types; missing columns; `value` is not numeric; `lags` contains negative or non-whole numbers; an output column name already exists.

See Also

`complete_daily_grid()`, `build_moving_average()`

Examples

```
df <- data.frame(
  site  = rep(c("A", "B"), each = 4),
  date  = rep(seq(as.Date("2024-05-01"), by = "day", length.out = 4), 2),
  count = c(10, 20, 30, 40, 100, 200, 300, 400)
)

apply_lag(df, value = "count", lags = 0:2, date = "date", by = "site")
```

---

assign_iso_week *Assign ISO 8601 week and year*

Description

`assign_iso_week()` appends the ISO 8601 week number and ISO year for each date. It is a row-wise mapping: no grouping, no temporal indexing, no requirement that the grid be complete.

Usage

```
assign_iso_week(data, date, week_col = "iso_week", year_col = "iso_year")
```

Arguments

`data`	A `data.frame`.
`date`	A single string giving the name of a `Date` column in `data`.
`week_col`	Name of the ISO week column to append. Default `"iso_week"`.
`year_col`	Name of the ISO year column to append. Default `"iso_year"`.

Details

The ISO year is not the calendar year, and this matters. Late-December and early-January dates routinely belong to an ISO week owned by the adjacent year:

Date	Weekday	ISO
2019-12-30	Monday	2020-W01
2020-12-31	Thursday	2020-W53
2021-01-01	Friday	2020-W53
2023-01-01	Sunday	2022-W52

Consequently, **week number alone is not a valid grouping key.** Grouping a multi-year series by `iso_week` pools week 1 of every year together, and pools the days either side of a New Year boundary into different weeks than the calendar suggests. Always group by `c(iso_year, iso_week)`. Downstream aggregation functions in SporeLag require both columns for this reason.

ISO weeks are computed in base R via the nearest-Thursday rule rather than `strftime()`'s `%V`/`%G` format codes, whose behaviour has historically varied by platform. Output is therefore identical on every operating system.

Value

`data` with two integer columns appended: `week_col` (1–53) and `year_col`. Row order and all input columns are unchanged.

Errors

`sporelag_error_bad_input` `data`, `date`, `week_col`, or `year_col` do not match the documented types; `date` is not a `Date` column; or an output column name already exists in `data`.

`sporelag_error_missing_date` `date` contains `NA`.

See Also

`assign_season()`

Examples

```
df <- data.frame(
  date = as.Date(c("2019-12-30", "2020-12-31", "2021-01-01")),
  count = c(1, 2, 3)
)
assign_iso_week(df, date = "date")
```

`assign_season` *Assign a season label*

Description

`assign_season()` appends a season label to each date. The definition is an explicit argument with no silent default behaviour, because "season" means materially different things in different literatures.

Usage

```
assign_season(
  data,
  date,
  definition = c("meteorological", "astronomical", "custom"),
  hemisphere = c("northern", "southern"),
  breaks = NULL,
  season_col = "season"
)
```

Arguments

`data`	A `data.frame`.
`date`	A single string giving the name of a `Date` column in `data`.
`definition`	One of `"meteorological"` (default), `"astronomical"`, or `"custom"`.
`hemisphere`	One of `"northern"` (default) or `"southern"`. Ignored when `definition = "custom"`, since custom breaks are absolute.
`breaks`	Required when `definition = "custom"`. A named character vector of `"MM-DD"` season start dates, e.g. `c(Ragweed = "08-15", Grass = "05-01", Tree = "02-15", Dormant = "11-01")`. Order does not matter; breaks are sorted internally.
`season_col`	Name of the column to append. Default `"season"`.

Details

`definition = "meteorological"` (the default) uses whole calendar months — Northern Hemisphere: Winter = Dec–Feb, Spring = Mar–May, Summer = Jun–Aug, Autumn = Sep–Nov. Boundaries are exact and leap-year stable.

`definition = "astronomical"` uses **fixed conventional dates** (Mar 20, Jun 21, Sep 22, Dec 21). True equinoxes and solstices drift by up to a day or two across years and depend on time zone; computing them exactly requires an ephemeris, which is out of scope for a zero-dependency package. If your analysis is sensitive to a one-day boundary shift, do not use this option — supply exact dates via `definition = "custom"`.

`definition = "custom"` takes `breaks`: a named character vector of `"MM-DD"` start dates, where the names are the season labels. Each season runs from its start date until the day before the next,

wrapping the year boundary. This is the option to use for **pollen seasons**, which are taxon- and region-specific and bear no necessary relation to the calendar.

Epidemiologic caution. In aeroallergen research, "season" usually means *pollen season* — a taxon-specific exposure window, often defined by a cumulative-count threshold, that varies by region and by year. It is not the astronomical or meteorological calendar. Using a calendar season as a proxy for a pollen season will misclassify exposure windows. The default here is a calendar convenience, not an exposure definition; state which you used in any analysis.

The returned column is an **ordered factor** whose levels follow the chronological cycle, so its behaviour in a model matrix is predictable. The reference level is the first level; set it explicitly with `stats::relevel()` if your model requires a different baseline.

Value

`data` with an ordered factor column appended. Row order and all input columns are unchanged.

Errors

`sporelag_error_bad_input` Bad types; `date` is not a `Date` column; `season_col` already exists; `breaks` missing, unnamed, duplicated, or not in `"MM-DD"` form.

`sporelag_error_missing_date` `date` contains `NA`.

See Also

`assign_iso_week()`

Examples

```
df <- data.frame(date = as.Date(c("2024-01-15", "2024-04-15", "2024-07-15")))

assign_season(df, date = "date")
assign_season(df, date = "date", hemisphere = "southern")

# A taxon-specific pollen season
assign_season(
  df,
  date = "date",
  definition = "custom",
  breaks = c(Dormant = "11-01", Tree = "02-15", Grass = "05-01",
             Ragweed = "08-15")
)
```

`build_moving_average` *Build moving-average exposure windows*

Description

`build_moving_average()` appends one column per requested window, containing the mean exposure over that window, computed within each group.

Usage

```
build_moving_average(
  data,
  value,
  window,
  date,
  by = character(),
  align = c("right", "center"),
  min_obs = window
)
```

Arguments

`data`	A `data.frame` on a complete daily grid.
`value`	A single string naming a numeric exposure column.
`window`	A vector of positive whole numbers, e.g. `c(3, 7)`.
`date`	A single string naming a `Date` column.
`by`	A character vector of grouping column names. Default `character()`.
`align`	`"right"` (default, trailing) or `"center"`. See Details.
`min_obs`	Minimum non-missing observations required within a window. Defaults to `window` (no missingness tolerated). Recycled if `window` has length > 1.

Details

Like `apply_lag()`, this function indexes by position and therefore **requires a complete daily grid**; it raises `sporelag_error_gaps` otherwise. Call `complete_daily_grid()` first.

Documented defaults — each of these changes what the variable means:

`align = "right"` **(default)** The window is **trailing** and **inclusive of the current day**: `window = 3` averages today and the two preceding days. This is the alignment appropriate for exposure→outcome inference.

`align = "center"` The window is symmetric around the current day and therefore **incorporates future exposure**. If the outcome is measured on the current day, a centred window conditions on information that did not exist at the time — it can induce apparent associations with no causal interpretation. It is provided for smoothing and descriptive display, not for exposure construction. Requires an odd `window`.

Edge windows are `NA` A window that extends past the start (or, when centred, either end) of a group's series is incomplete by construction and returns `NA`. The first `window - 1` days of each trailing series are `NA`. They are not back-filled with a shorter average, because a "7-day mean" computed from 2 days is not a 7-day mean.

`min_obs = window` **(default)** The minimum number of **non-missing** observations required *within* an otherwise complete window. At the default, any `NA` in the window yields `NA`. Lowering it (e.g. `min_obs = 5` with `window = 7`) tolerates missing days and averages over those present — which shrinks variance and can bias a lag–response estimate toward the null. Lower it deliberately, and report that you did.

Value

`data` with one `double` column appended per window, named `{value}_ma{k}`. Row order and all input columns are unchanged.

Errors

`sporelag_error_gaps` The daily grid is incomplete within at least one group.

`sporelag_error_bad_input` Bad types; `value` is not numeric; `window` is not a positive whole number; `min_obs` exceeds `window`; `align = "center"` with an even `window`; an output column already exists.

See Also

`complete_daily_grid()`, `apply_lag()`

Examples

```
df <- data.frame(
  site  = rep(c("A", "B"), each = 5),
  date  = rep(seq(as.Date("2024-05-01"), by = "day", length.out = 5), 2),
  count = c(10, 20, 30, 40, 50, 100, 200, 300, 400, 500)
)

build_moving_average(df, value = "count", window = 3, date = "date",
                     by = "site")
```

`complete_daily_grid` *Build a complete daily grid*

Description

`complete_daily_grid()` fills gaps in a daily time series so that every calendar day between the minimum and maximum date (within each group) is represented by exactly one row. Newly inserted rows carry `NA` for every column except `date` and the columns named in `by`.

Usage

```
complete_daily_grid(data, date, by = character())
```

Arguments

data	A data.frame.
date	A single string giving the name of a Date column in data.
by	A character vector of column names in data to group by. Defaults to character(), meaning no grouping (the whole data.frame is treated as a single daily series). Gaps are filled, and dates are checked for duplicates, independently within each group.

Details

Functions in SporeLag that index by time (lags, moving averages) assume the day-to-day step is exactly one day with no repeats and no gaps within a group. Rather than silently shifting values by row position when that assumption fails, those functions raise a classed error. This function is the documented mechanism for producing that guarantee before calling them.

complete_daily_grid() never guesses at missing values: inserted rows get NA, and it is the caller's responsibility to decide whether and how to impute them.

Value

A data.frame with the same columns as data, in the same order, with one row per calendar day between min(date) and max(date) within each group. Rows are sorted by by, then date. No columns are added or removed; this function only inserts rows.

Errors

sporelag_error_bad_input data, date, or by do not match the documented types, or name columns not present in data.

sporelag_error_duplicate_dates The same date appears more than once within a single group. A grid cannot be built unambiguously, so the function errors instead of silently choosing a row.

Examples

```
df <- data.frame(
  site = c("A", "A", "A", "B", "B"),
  date = as.Date(c(
    "2024-01-01", "2024-01-02", "2024-01-04",
    "2024-01-01", "2024-01-03"
  )),
  count = c(1, 2, 4, 10, 30)
)
complete_daily_grid(df, date = "date", by = "site")
```

`impute_weekly_mean` *Impute missing daily values with the ISO-week mean*

Description

`impute_weekly_mean()` fills missing daily exposure values with the mean of the observed values in the same ISO week, within group. It appends the completed series and a flag marking which values were imputed. **The input column is never modified.**

Usage

```
impute_weekly_mean(
  data,
  value,
  by = character(),
  week_col = "iso_week",
  year_col = "iso_year",
  min_obs = 1L
)
```

Arguments

`data`	A `data.frame` containing ISO week and year columns.
`value`	A single string naming a numeric exposure column.
`by`	A character vector of grouping column names. Default `character()`.
`week_col`	Name of the ISO week column. Default `"iso_week"`.
`year_col`	Name of the ISO year column. Default `"iso_year"`.
`min_obs`	Minimum observed days required in a week for its mean to be used. Default `1`.

Details

Requires the ISO week and year columns produced by `assign_iso_week()`. Typically it also follows `complete_daily_grid()`, since a day that is absent from the data entirely cannot be imputed — it must first exist as a row with an `NA` value.

The canonical pipeline is:

```
data |>
  complete_daily_grid(date = "date", by = "site") |>
  assign_iso_week(date = "date") |>
  impute_weekly_mean(value = "count", by = "site")
```

Grouping is by `c(by, year_col, week_col)` — never by week alone. ISO week 1 recurs every year, and the ISO year of a late-December date is often the *following* calendar year. Grouping on week number alone would pool observations from different years into a single mean. See `assign_iso_week()`.

Documented defaults:

`min_obs = 1` The minimum number of observed days required in a week before its mean is used. At the default, a single observed day is enough to fill the other six. Raise it (e.g. `min_obs = 4`) if you are unwilling to extrapolate a week from one observation; weeks below the threshold are left `NA` and flagged `FALSE`.

Weeks with no observations stay `NA` Nothing is carried across a week boundary. There is no interpolation between weeks, no last-observation-carried-forward, and no global mean fallback.

Output is `double` Even when `value` is an integer count. A mean is not a count, and rounding an imputed 12.5 to 12 or 13 would fabricate precision. If your model needs integers, round explicitly and say so.

Epidemiologic caution. Weekly-mean imputation replaces day-to-day variation with a constant, which *shrinks the variance of the exposure*. In a lag–response model, an exposure with attenuated variance typically produces effect estimates biased *toward the null*, and standard errors that do not reflect the true uncertainty (the imputed values are treated as though observed). The `{value}_imputed_flag` column exists so that you can quantify this: report the proportion imputed, and re-run the analysis on complete cases as a sensitivity check. This function makes imputation convenient; it does not make it innocuous.

Value

`data` with two columns appended:

`{value}_imputed` `double`. The completed series.

`{value}_imputed_flag` `logical`. `TRUE` where a value was filled.

Row order and all input columns, including `value`, are unchanged.

Errors

`sporelag_error_missing_col` `week_col` or `year_col` is absent. Call `assign_iso_week()` first.

`sporelag_error_bad_input` Bad types; `value` is not numeric; `min_obs` is not a positive whole number; an output column already exists.

See Also

`assign_iso_week()`, `complete_daily_grid()`

Examples

```
df <- data.frame(
  site  = rep("A", 7),
  date  = seq(as.Date("2024-05-06"), by = "day", length.out = 7),  # Mon-Sun
  count = c(10, NA, 30, NA, 50, 60, NA)
)

df |>
  assign_iso_week(date = "date") |>
  impute_weekly_mean(value = "count", by = "site")
```

`pollen_demo` *Synthetic daily pollen counts from two monitoring sites*

Description

A **synthetic** dataset of daily airborne pollen concentrations for a single spring tree-pollen season at two hypothetical monitoring stations. It exists to demonstrate and test SporeLag and is used throughout `vignette("getting-started")`.

Usage

```
pollen_demo
```

Format

A data frame with three columns:

site `character`. Monitoring station: `"North"` or `"South"`.

date `Date`. Calendar day, spanning 2024-02-15 to 2024-06-30.

count `integer`. Pollen concentration in grains per cubic metre. `NA` where sampling failed on a day that was otherwise monitored.

Details

The data deliberately contain **both** of the defects SporeLag is designed to handle, because they are different problems and are frequently confused:

Gaps Calendar days with **no row at all**, representing periods when the sampler was offline: 2024-03-18 to 2024-03-22 at North, and 2024-04-01 to 2024-04-03 at South. A gap is invisible to `is.na()` — the row simply is not there. Gaps must be closed with `complete_daily_grid()` before any lag or moving average is computed, or the shift will be by row position rather than by time.

Missing values Days that **are** present but whose `count` is `NA`, representing failed samples on otherwise-monitored days (roughly 6% of days at each site). These are candidates for `impute_weekly_mean()`.

The South gap falls at that site's seasonal peak — the point at which losing data does the most damage to an exposure estimate, and where a naive positional lag would silently produce the largest misalignment.

Counts were drawn from a negative binomial distribution around a Gaussian seasonal curve, so they are overdispersed in the way real aeroallergen counts are. The generating script is `data-raw/make_pollen_demo.R` and is fully deterministic.

Source

Simulated. **These are not real surveillance data** and must not be used for any substantive inference about pollen exposure. See `data-raw/make_pollen_demo.R` for the generating code.

See Also

complete_daily_grid(), impute_weekly_mean()

Examples

```
str(pollen_demo)

# Gaps are absent rows, not NA values -- is.na() cannot see them:
table(pollen_demo$site)          # neither site has a full 137 days
sum(is.na(pollen_demo$count))    # the NAs are a separate problem

# The canonical pipeline:
pollen_demo |>
  complete_daily_grid(date = "date", by = "site") |>
  assign_iso_week(date = "date") |>
  impute_weekly_mean(value = "count", by = "site") |>
  apply_lag(value = "count_imputed", lags = 0:2, date = "date", by = "site") |>
  head()
```

Index

∗ **datasets**
pollen_demo, 12

apply_lag, 2
apply_lag(), *7, 8*
assign_iso_week, 3
assign_iso_week(), *6*, *10, 11*
assign_season, 5
assign_season(), *4*

build_moving_average, 7
build_moving_average(), *3*

complete_daily_grid, 8
complete_daily_grid(), *2, 3*, *7, 8*, *10–13*

impute_weekly_mean, 10
impute_weekly_mean(), *12, 13*

pollen_demo, 12

stats::relevel(), *6*