Package 'tbl.now'

September 21, 2026

Type Package

Title Tidy Data and Workflow Layer for Epidemic Nowcasting

Version 1.0.0

Description Defines tidy data structures and package-agnostic workflows for epidemiological nowcasting. The 'tbl_now' class records event, report, and revision dates alongside strata, covariates, censoring, and reporting-delay metadata while remaining compatible with 'dplyr'. Tools support validation, manipulation, diagnostics, visualization, format conversion, retrospective evaluation, and multiple modelling engines. The 'tbl_nowcast' class standardizes probabilistic predictions for plotting, scoring, comparison, and ensembling.

License MIT + file LICENSE

Encoding UTF-8

LazyData true

LazyDataCompression xz

Depends R (>= 4.2.0)

Suggests almanac, baselinenowcast (>= 0.2.1), diseasenowcasting (>= 2.4.0), data.table, epidist (>= 0.4.1), EpiNow2 (>= 1.9.0), epinowcast (>= 0.7.0), knitr, modifiedmk (>= 1.6.0), NobBS (>= 1.1.1), patchwork, plotly, rmarkdown, scoringutils (>= 2.0.0), surveillance (>= 1.26.1), testthat (>= 3.0.0), tsibble, withr

Additional_repositories https://epinowcast.r-universe.dev, https://davisvaughan.r-universe.dev, https://rodrigozepeda.r-universe.dev

Config/testthat/edition 3

Config/testthat/parallel false

Imports cli, generics, grid, dplyr, ggplot2, lifecycle, lubridate, methods, pillar, rlang, S7, scales, stats, tibble, tidyr, tidyselect, utils

URL https://rodrigozepeda.github.io/tbl.now/, https://github.com/RodrigoZepeda/tbl.now

BugReports https://github.com/RodrigoZepeda/tbl.now/issues

VignetteBuilder knitr

Config/Needs/website rmarkdown

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Rodrigo Zepeda-Tello [aut, cre] (ORCID:
<https://orcid.org/0000-0003-4471-5270>),
Rami Yaari [aut] (ORCID: <https://orcid.org/0000-0002-8808-8937>),
Matteo Perini [aut] (ORCID: <https://orcid.org/0000-0002-9465-6216>),
Teresa Yamana [ctb] (ORCID: <https://orcid.org/0000-0001-8349-3151>),
Jeffrey Shaman [ctb] (ORCID: <https://orcid.org/0000-0002-7216-7809>),
Columbia University in the City of New York [cph, fnd]

Maintainer Rodrigo Zepeda-Tello <rzepeda17@gmail.com>

Repository CRAN

Date/Publication 2026-09-21 21:10:02 UTC

Contents

---

add *Set, change and remove the attributes of a* `tbl_now`

Description

[Stable]

A `tbl_now()` remembers which of its columns are the event date, the report date, the strata, and so on. These functions edit that memory after the object has been built – useful when new columns appear part-way through a pipeline, or when you want to nowcast the same data broken down a different way.

There are three verbs, and they differ only in what they do to what is already recorded:

- `add_*()` keeps what is there and adds to it. `add_strata(x, age_group)` on an object already stratified by gender leaves you stratified by both.
- `change_*()` replaces it. `change_strata(x, age_group)` on the same object leaves you stratified by age group *only*.
- `remove_*()` takes it away. `remove_all_strata(x)` forgets every stratum; `remove_strata(x, gender)` forgets just that one.

Nothing here touches the data itself. Renaming an attribute does not rename, create or delete a column – it only changes which existing column the object treats as playing that role.

Usage

```
change_now(x, now = NULL, verbose = TRUE)

update_now(x, verbose = TRUE)

change_event_date(x, event_date)

change_report_date(x, report_date)

change_case_count(x, case_count)
```

```
change_is_censored_report(x, is_censored_report)

remove_is_censored_report(x)

add_is_censored_report(x, is_censored_report)

change_strata(x, ..., warn_now = TRUE, warn_non_uniqueness = TRUE)

remove_strata(x, ...)

add_strata(x, ...)

remove_all_strata(x)

change_covariates(x, ..., warn_now = TRUE, warn_non_uniqueness = TRUE)

remove_covariates(x, ...)

add_covariates(x, ...)

remove_all_covariates(x)

replace_temporal_effects(x, t_effects)

remove_temporal_effects(x)

change_is_censored_revision(x, is_censored_revision)

add_is_censored_revision(x, is_censored_revision)

remove_is_censored_revision(x)

add_revision_date(
  x,
  revision_date,
  revision_type = NULL,
  revision_units = "auto",
  revision_levels = NULL
)

change_revision_date(
  x,
  revision_date,
  revision_type = NULL,
  revision_units = "auto",
  revision_levels = NULL
)
```

```
remove_revision_date(x)
```

Arguments

x	A tbl_now object.
now	(optional) Date or NULL (default). The date that is considered the now-cast. If no now is given then the function automatically uses the last event_date.
verbose	(optional) Logical. Whether to throw a message. Default = TRUE.
event_date	tidy-select name of the column containing the event date. Optional when delay is provided together with report_date; the event date will be computed as report_date - delay.
report_date	tidy-select name of the column containing the report date. Optional when delay is provided together with event_date; the report date will be computed as event_date + delay.
case_count	(optional) tidy-select or NULL Name of the column with the case counts if data_type is "count-incidence" or "count-cumulative".
is_censored_report	(optional) tidy-select or NULL (default). The name of a column containing either TRUE or FALSE indicating whether the report_date is correctly specified or corresponds to a batch and thus is censored. In other words, if the report_date is accurately measured set is_censored_report = FALSE but if the report_date corresponds to an error and is only an upper bound of the real report date set is_censored_report = TRUE.
...	tidy-select columns for the attribute being set. For strata and covariates this may name several columns at once.
warn_now	Boolean. Whether to warn if now falls before the last report date, or unreasonably far into the future.
warn_non_uniqueness	(optional) Logical. Whether to throw a warning if data has several rows on the same full key: the event, report and (when declared) revision dates, the revision type, the strata, the covariates and the censoring flags. Rows carrying an NA anywhere in that key – a pending case has no revision date – are left out of the comparison, because an unobserved value cannot be shown to equal another.
t_effects	(optional) Either NULL (default), a temporal_effects() object or a character vector with the names of the columns containing the temporal effects.
is_censored_revision	(optional) tidy-select or NULL (default). The revision-axis counterpart of is_censored_report: the name of a logical column marking rows whose **revision delay** is a bound rather than a measurement. Requires a revision_date. See censor_revision_delays_above().
revision_date	(optional) tidy-select column holding a **third** date: the day the report was resolved. Influenza is the picture to keep in mind – symptoms begin (the event), the patient sees a doctor (the report), and days later a swab comes back. The assumed timeline is event_date <= report_date <= revision_date <= now. Leave NULL (the default) for the usual two-date object. See add_revision_date().

`revision_type` (optional) tidy-select column saying what the resolution *was*: `"confirmed"`, `"retracted"` (it was reported, but it is not a case after all), `"pending"` or `NA`. `"pending"` **means reported and still waiting**, so it carries no revision date – which is a different thing from a result that was never recorded (`NA`). A revision date with no type warns rather than guessing, because a date alone cannot say whether the case was confirmed or retracted.

`revision_units` (optional) Character. Either `"auto"` (default), `"days"`, `"weeks"`, `"months"`, `"years"` or `"numeric"` – the grid the revision date lives on, resolved the same way as `report_units`.

`revision_levels` (optional) `NULL` (default) or a **named** character vector translating the labels in `revision_type` into the canonical outcomes, for data that was not recorded in English: `c(confirmado = "confirmed", retractado = "retracted", pendiente = "pending")`. The names are the labels in your data, the values are the canonical ones. The column is rewritten to the canonical values and the dictionary is kept as an attribute, readable with `get_revision_levels()`. Only `"confirmed"`, `"retracted"`, `"pending"` and `NA` are ever stored.

Details

Columns are chosen with tidy-select, so a bare column name works, and so do the helpers `dplyr::starts_with()`, `dplyr::all_of()` and `dplyr::where()`. See `dplyr::select()` for the full set.

`update_now()` deserves a note of its own. The now of a nowcast is the day you are standing on, and it does **not** move when you filter the data – an object filtered down to 1992 still believes now is 2010, which is usually not what you want. `update_now()` resets it to the latest date actually present:

```
data(denguedat)
ndata <- tbl_now(denguedat,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

# `now` is in 2010, because the data runs that far.
get_now(ndata)
#> [1] "2010-12-20"

# Filtering the data does not move it ...
ndata_1992 <- ndata |>
  dplyr::filter(
    onset_week <= as.Date("1992/01/01") & report_week <= as.Date("1992/01/01")
  )
get_now(ndata_1992)
#> [1] "2010-12-20"

# ... but `update_now()` does.
get_now(update_now(ndata_1992))
#> [1] "1991-12-30"
```

Value

A tbl_now object with the attribute updated. The data are returned unchanged; only what the object records about itself differs.

The revision process, the optional third date

add_revision_date(), change_revision_date() and remove_revision_date() set the **third** date a surveillance record can carry: after the event happened and after it was reported, somebody decided whether it was real. For influenza that is the laboratory result – and it can come back negative, in which case the case is *retracted* rather than confirmed.

Attaching one is the only verb on this page that changes more than a name:

- now **moves.** A revision is an observation, so the as-of moment becomes the latest of the report and revision dates. Revision refuses an object whose now falls before a revision that has already happened.
- **Two columns appear.** .revision_num is the date on the same numeric anchor as .event_num/.report_num; .revision_delay is the time from report to resolution. Both are protected, like .delay.
- **Counting gains a dimension.** to_count() groups by the revision date and outcome as well, so a confirmed and a retracted case on the same (event, report) pair stay separate rather than being summed together.
- **The timeline is checked.** event_date <= report_date <= revision_date; rows that break it are warned about, not silently accepted.

A date on its own cannot say whether the test came back positive or negative, so leaving revision_type out gives every dated row NA and warns.

Two optional pieces travel with the third date. revision_levels is a named dictionary translating the labels in your data into the four values revision_type may hold – c(confirmado = "confirmed", ...) – so the recoding happens once rather than in every script. And add_is_censored_revision() names a logical column marking rows whose *revision delay* is a bound rather than a measurement, the revision-axis twin of add_is_censored_report(); censor_revision_delays_above() sets it for you.

change_now() is revision-aware in both directions. Moving now forward does nothing to the data; moving it **backwards**, which is how a backtest asks what was known at an earlier date, returns every revision dated after that moment to "pending" and masks its date. A resolution that has not happened yet is not a resolution.

See Also

tbl_now() for setting these when the object is first built; the getters for reading them back; tbl_now_attributes() to list them all at once; add_temporal_effects() and temporal_effects() for calendar structure; update() for appending new rows rather than editing attributes.

Examples

```
data(denguedat)
# These verbs only touch what the object records about itself, never the
# rows, so a couple of years stands in for the full twenty-year series.
recent <- denguedat[denguedat$onset_week >= as.Date("2009-01-01"), ]
```

```
ndata <- tbl_now(recent,
  event_date = onset_week,
  report_date = report_week,
  strata = gender,
  verbose = FALSE
)

## ---- Strata: add, change, remove -----------------------------------

ndata$age_group <- sample(c("<18", "18-60", "60+"), nrow(ndata), replace = TRUE)

## `add_strata()` keeps gender and adds age group.
get_strata(add_strata(ndata, age_group))

## `change_strata()` replaces gender with age group.
get_strata(change_strata(ndata, age_group))

## `remove_strata()` drops one; `remove_all_strata()` drops the lot.
get_strata(remove_strata(ndata, gender))
get_strata(remove_all_strata(add_strata(ndata, age_group)))

## ---- Covariates behave the same way --------------------------------

# Covariates influence the nowcast but are not of interest in themselves.
# They should be values known at the fitted object's `now`; do not let
# future realized covariates leak into a backtest snapshot.
ndata$temperature <- rnorm(nrow(ndata), 25, 4)
ndata$humidity <- rbeta(nrow(ndata), 0.6, 0.4)
ndata <- ndata |> add_covariates(temperature, humidity)
get_covariates(ndata)

ndata |>
  remove_covariates(humidity) |>
  get_covariates()

## ---- Pointing an attribute at a different column -------------------

## Suppose onset was recorded a day late and you correct it. `change_event_date()`
# tells the object to use the corrected column instead.
ndata$corrected_onset <- ndata$onset_week - lubridate::days(1)
ndata <- ndata |> change_event_date(corrected_onset)
get_event_date(ndata)

## ---- The censoring indicators ---------------------------------------

## TRUE means the report date is only an upper bound (e.g. a backlog dump).
ndata$is_censored_report <- FALSE
ndata <- ndata |> add_is_censored_report(is_censored_report)
get_is_censored_report(ndata)
ndata <- remove_is_censored_report(ndata)

## ---- `now` ----------------------------------------------------------
```

```
# Set it by hand ...
get_now(change_now(ndata, now = as.Date("2011-01-01")))

# ... or snap it back to the latest date actually observed.
get_now(update_now(ndata))

## ---- Count data: which column holds the counts ------------------------

counts <- to_count(ndata, to = "count-incidence")
counts |>
  dplyr::mutate(inflated = round(1.15 * n)) |>
  change_case_count(inflated) |>
  get_case_count()

## ---- The revision process, the optional third date -----------------

data(covid_us)
covid <- covid_us |>
  dplyr::filter(onset_dt >= as.Date("2020-12-01")) |>
  tbl_now(
    event_date = onset_dt, report_date = pos_spec_dt,
    case_count = n, data_type = "count-incidence",
    verbose = FALSE, warn_non_uniqueness = FALSE
  )

## Onset -> positive specimen -> registration at CDC. A date alone cannot say
# how the case resolved, so this warns until an outcome column is supplied.
covid <- suppressWarnings(add_revision_date(covid, cdc_report_dt))
get_revision_date(covid)

## CDC's own labels are not this package's four, which is what
# `revision_levels` translates.
covid <- change_revision_date(covid, cdc_report_dt,
  revision_type = current_status,
  revision_levels = c(
    "Laboratory-confirmed case" = "confirmed", "Probable Case" = "pending"
  )
)
table(covid[[get_revision_type(covid)]])
get_revision_levels(covid)

## A revision delay you refuse to believe is a bound, not a measurement.
covid <- censor_revision_delays_above(covid, 45, verbose = FALSE)
get_is_censored_revision(covid)

## Dropping the third date leaves an ordinary two-date object.
has_revision(remove_revision_date(covid))

## ---- Temporal effects -------------------------------------------------

# Recorded lazily: `replace_*` swaps the specification, `remove_*` forgets it.
ndata <- ndata |>
  add_temporal_effects(
```

```
    t_effects = temporal_effects(week_of_year = TRUE, month_of_year = TRUE)
  )
get_temporal_effects(ndata)

ndata |>
  replace_temporal_effects(t_effects = temporal_effects(seasons = 52)) |>
  get_temporal_effects()

ndata |>
  remove_temporal_effects() |>
  get_temporal_effects()
```

`add_temporal_effects` *Attach calendar effects to a* `tbl_now`*, and turn them into columns*

Description

[Stable]

These are the second and third steps of using calendar structure in a nowcast. First you write down which patterns you want with `temporal_effects()`; then:

- `add_temporal_effects()` **records** that request on the object. Nothing is computed and no columns appear – the specification is stored lazily, so it survives filtering and joining without going stale.
- `compute_temporal_effects()` **materialises** it, building one column per effect from the object's dates.

The split matters because the columns depend on the data. If you computed them first and then filtered, or changed the event-date column, the columns would silently describe the wrong rows. Recording the request and computing it at the end avoids that.

Call `add_temporal_effects()` more than once to accumulate several specifications on the same object; `compute_temporal_effects()` builds all of them.

Usage

```
add_temporal_effects(x, t_effects = NULL, overwrite = FALSE, ...)

## S3 method for class 'data.frame'
add_temporal_effects(
  x,
  t_effects = NULL,
  overwrite = FALSE,
  ...,
  date_col = NULL,
  numeric_col = NULL,
  name_prefix = paste0(".", date_col),
  weekend_days = c("Sat", "Sun"),
```

```
  units = "days"
)

## S3 method for class 'tbl_now'
add_temporal_effects(
  x,
  t_effects = NULL,
  overwrite = FALSE,
  ...,
  date_type = "event_date",
  weekend_days = c("Sat", "Sun")
)

compute_temporal_effects(x, overwrite = FALSE)
```

Arguments

x	A tbl_now object or a data.frame.
t_effects	A temporal_effects() object codifying the temporal effects to be used.
overwrite	Logical. When TRUE, columns that already exist are overwritten. When FALSE (the default) an existing column of the same name is an error, so an accidental second computation cannot silently replace your data.
...	Additional arguments (unused)
date_col	The column which contains the <Date> values from which effects will be calculated. This applies to all temporal_effects except for seasonal.
numeric_col	The column which contains the values from which the seasonal effects will be calculated. This applies only to seasonal effects. For date-related effects (such as month or day of the week) use date_col.
name_prefix	Character. Prefix for the names of the created columns.
weekend_days	A character or numeric vector defining which days count as the weekend. Defaults to Saturday and Sunday. • Character: day names or abbreviations, case-insensitive – c("Mon", "Tuesday", "wed", ...). English names always work, whatever the session's locale is, and so do the names of the current locale, with or without their accents – under LC_TIME = "es_ES" both c("Sat", "Sun") and c("sáb", "dom") mean the weekend. • Numeric: integers 1-7 in lubridate::wday() numbering with week_start = 1, so **1 = Monday** and 7 = Sunday.
units	Character. The time units date_col is measured in: "days" (the default), "weeks", "months" or "years". It only affects the **holiday** column, which on a grid coarser than days becomes the *share* of the period's days that are holidays rather than a 0/1 indicator. For a tbl_now, compute_temporal_effects() reads this off the object.
date_type	One of event_date (default), report_date, or revision_date to add temporal effects to that column. revision_date requires a tbl_now with a revision process.

Value

`add_temporal_effects()` returns the object with the specification recorded. For a `tbl_now` **no columns are added**; for a plain `data.frame`, which has nowhere to record a specification, the columns are computed immediately.

`compute_temporal_effects()` returns a `tbl_now` with one column per effect appended. The specification is kept, so it still prints; the names of the columns just created are available from get_temporal_effect_cols().

See Also

`temporal_effects()` to build the specification, and for what each effect means; replace_temporal_effects() and remove_temporal_effects() to swap or drop it; get_temporal_effects() and get_temporal_effect_cols() to read back the request and the columns; calendar_effect_plots to see the patterns; `add()` for the other attribute setters.

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat,
  event_date = "onset_week",
  report_date = "report_week",
  strata = "gender",
  verbose = FALSE
)

# Step 1-2: say you want a week-of-year effect, and record it.
dengue <- dengue |>
  add_temporal_effects(t_effects = temporal_effects(week_of_year = TRUE))

# The request is stored, but no column has been built yet.
get_temporal_effects(dengue)
get_temporal_effect_cols(dengue)

# Step 3: materialise it.
computed <- compute_temporal_effects(dengue)
get_temporal_effect_cols(computed)
head(computed[[get_temporal_effect_cols(computed)[1]]])

# Specifications accumulate, so you can add a second pattern ...
both <- dengue |>
  add_temporal_effects(t_effects = temporal_effects(month_of_year = TRUE))
get_temporal_effect_cols(compute_temporal_effects(both))

# ... swap the whole specification for another ...
dengue |>
  replace_temporal_effects(t_effects = temporal_effects(seasons = 52)) |>
  get_temporal_effects()

# ... or forget it entirely.
dengue |>
  remove_temporal_effects() |>
```

```
get_temporal_effects()
```

`aggregate_time_units` *Coarsen a* `tbl_now` *onto a bigger time unit*

Description

[Stable]

Daily surveillance data is often too sparse to nowcast: most (event date, report date) cells hold a zero or a one, and the delay distribution is mostly noise. The usual fix is to work in weeks instead. `aggregate_time_units()` does that in one call – it moves every date onto the coarser grid, adds the counts up, and returns a `tbl_now` that *knows* it is now weekly, so `.delay`, the converters and the models all count in weeks from then on.

Usage

```
aggregate_time_units(
  x,
  to = "weeks",
  axes = "all",
  label = c("start", "end"),
  align_on_day = 7,
  type = "epi",
  verbose = TRUE
)
```

Arguments

`x`	A `tbl_now()` object.
`to`	Character. The unit to aggregate **to**: `"weeks"` (the default), `"months"`, `"years"` or `"days"`. It must be at least as coarse as the units of every axis being aggregated – there is no un-aggregating, so asking a weekly object for `"days"` is an error rather than a guess.
`axes`	Character. Which time axes to aggregate: `"all"` (the default, meaning the event and report axes plus the revision axis when there is one), or any of `"event"`, `"report"` and `"revision"`. The unit they are aggregated to is `to`.
`label`	Character. Which end of the period names it: `"start"` (the default – the Sunday of the epi week, the first of the month) or `"end"` (its last day). It only changes the *labels*, never which rows are pooled together, but see *Aggregating one axis only* for when it matters.
`align_on_day, type`	Passed to `align_weeks()` when `to = "weeks"`: the weekday to snap to (ISO numbering, **1 = Monday**, default 7 = Sunday) and the week convention, `"epi"` (default) or `"iso"`.
`verbose`	Logical. Whether to report what was aggregated. Default `TRUE`.

Details

Each date is replaced by the **start of the period it falls in**: the Sunday that begins its epidemiological week, the first of its month, the first of January of its year (`label = "end"` names the period by its last day instead). Weeks go through the same epi/ISO machinery `align_weeks()` uses, so `type` and `align_on_day` mean exactly what they mean there.

What happens to the rows depends on the data type:

- `linelist` – one row is still one case; only the dates move.
- `count-incidence` – rows that land in the same (event, report) cell are summed.
- `count-cumulative` – cumulative totals are **not** additive, so the series is de-accumulated to increments first, aggregated, and accumulated again on the new grid.

Two consequences worth knowing:

- **Weeks do not nest inside months.** Aggregating daily data to weeks and *then* to months is not the same as going straight to months: the second pass sees only the week's label, so a week beginning 31 December lands in December even though most of its cases happened in January. Aggregate once, to the unit you actually want.
- An `NA` count means *not yet observed*, so a period containing one has an unknown total, not a total that quietly leaves it out. Use `complete_zeroes()` first if the `NA`s are really zeroes.

Value

A `tbl_now` on the coarser grid, with `event_units`, `report_units` and `revision_units` updated for the axes that were aggregated, and `now` moved onto the new grid.

What happens to the temporal effects

Any temporal-effect **columns** that were materialised by `compute_temporal_effects()` are dropped, because a day-of-week term computed on daily dates is meaningless once those dates are weeks.

The lazy `temporal_effects()` specification moves onto the new grid with everything else, and the effects the new grid cannot express are **dropped from it** rather than silently rebuilt on dates that cannot carry them:

- `day_of_week`, `weekend`, `day_of_month`, `holiday_lags` and `weekend_lags` are properties of a *day*, so they survive only `to = "days"`.
- `week_of_year` survives `"weeks"`; `month_of_year` survives `"months"`.
- `seasons` **are rescaled.** A Fourier period is a length, not a position in the calendar, so it converts: a 365-day season becomes a 52.14-week one, and `seasons = 52, season_length = 7` becomes `seasons = 52` in weeks. A period that ends up two units or shorter is dropped – it is at or below the new grid's Nyquist limit, so it can no longer be told from a constant or from a longer wave.
- `holidays` **are kept.** On a coarser grid the holiday column stops being a 0/1 indicator and becomes the **share of the period's days that the calendar marks** – 1/7 for a week containing Christmas Day – which is the same number as the indicator when the period is one day long.

A specification with nothing left is removed. `verbose = TRUE` says what was dropped and what was rescaled.

Aggregating one axis only

`axes` exists because the two axes do not always move together: a system may record the day a specimen was taken but only publish weekly report batches. Two things follow:

- `tbl_now()` requires the report axis to be **at least as coarse** as the event axis, so aggregating only the event axis of a daily object is refused. Aggregate the report axis too, or use `axes = "all"`.
- A week named by the day it *starts* sits before every date inside it, so coarsening a **later** axis alone – the report against a daily event, or the revision against a daily report – with `label = "start"` produces negative delays, and `validate_tbl_now()` warns. Use `label = "end"`: a report that arrived somewhere in week W is known by the end of W, which is the honest bound. When the axes move together the labelling cancels out and either choice gives the same delays.

See Also

`align_weeks()`, which snaps weekly dates to a common weekday without changing the units or the counts; `to_count()` to change data type without touching the dates; `complete_zeroes()` for the cells the coarser grid still leaves empty; `tbl_now()`'s `units` argument to declare the units up front; `temporal_effects()` and `compute_temporal_effects()` for the specification this coarsens.

Examples

```
# A sparse daily line list: one or two cases a day.
df <- data.frame(
  onset = as.Date("2024-01-01") + c(0, 1, 3, 8, 9, 15),
  reported = as.Date("2024-01-01") + c(2, 2, 5, 9, 12, 16),
  sex = c("F", "M", "F", "M", "F", "M")
)
daily <- tbl_now(df,
  event_date = onset, report_date = reported, strata = sex,
  data_type = "linelist", units = "days", verbose = FALSE
)
get_event_units(daily)

# The same cases, on a weekly grid: every date moves to the Sunday that
# starts its epidemiological week, and the delays are whole weeks.
weekly <- aggregate_time_units(daily, to = "weeks", verbose = FALSE)
get_event_units(weekly)
weekly[[get_event_date(weekly)]]
weekly$.delay

# Count data is added up rather than merely relabelled.
counts <- to_count(daily, to = "count-incidence")
sum(counts$n)
sum(aggregate_time_units(counts, to = "months", verbose = FALSE)$n)

# A weekly grid cannot carry a day-of-week effect, so it is dropped from the
# specification; the 365-day season is rescaled to 52.14 weeks instead.
spec <- daily |>
  add_temporal_effects(
```

```
    temporal_effects(day_of_week = TRUE, seasons = c(7, 365))
  )
get_temporal_effects(aggregate_time_units(spec, to = "weeks", verbose = FALSE))
```

`align_weeks` *Put weekly data on a common weekday*

Description

[Stable]

Weekly surveillance data is rarely as tidy as it looks. The same series may be stamped with a Wednesday one year and a Thursday the next, or event dates may fall on a Sunday while reports fall on a Saturday. When that happens the delay between the two stops being a whole number of weeks – you get delays of 2.86 weeks – and most nowcasting models, which count in whole periods, either refuse the data or quietly round it.

`align_weeks()` snaps every date to the same weekday, so week differences come out as integers. `week_2_date()` solves the neighbouring problem: you have epiweek (or ISO week) *numbers* rather than dates, and need real dates to build a `tbl_now` from.

Usage

```
align_weeks(.data, align_on_day = 7, type = "epi", ...)

## S3 method for class 'data.frame'
align_weeks(
  .data,
  align_on_day = 7,
  type = "epi",
  ...,
  date_col,
  new_date_col = NULL
)

## S3 method for class 'tbl_now'
align_weeks(.data, align_on_day = 7, type = "epi", ...)

week_2_date(
  .data,
  week_col,
  year_col,
  align_on_day = 7,
  week_fun = lubridate::epiweek,
  year_fun = lubridate::epiyear,
  date_col_name = "date"
)
```

Arguments

.data	A data.frame, tibble or tbl_now.
align_on_day	Integer 1-7 giving the weekday to align to, in ISO numbering: **1 = Monday**, 2 = Tuesday, ..., **7 = Sunday**. This is lubridate::wday() with week_start = 1, the same convention is_weekday() uses. Defaults to 7 (Sunday), the start of an epidemiological week.
type	Either "epi" (default) or "iso", choosing whether week and year are read with lubridate::epiweek() or lubridate::isoweek().
...	Additional arguments passed to methods.
date_col	For the data.frame method, the tidy-select column holding the dates to align.
new_date_col	Name for the aligned column. Defaults to \{date_col\}_aligned.
week_col, year_col	For week_2_date(), the tidy-select columns holding the week number and the year.
week_fun, year_fun	For week_2_date(), the functions defining the week convention: lubridate::epiweek()/lubridate::(the default) or lubridate::isoweek()/lubridate::isoyear().
date_col_name	For week_2_date(), the name of the date column to create.

Details

Applied to a data.frame, align_weeks() adds an aligned copy of the column you name. Applied to a tbl_now, it aligns the event and report dates together and recomputes the delay, so the object stays coherent.

Epi weeks and ISO weeks disagree about where a year starts, so type picks which convention to use: "epi" uses lubridate::epiweek() / lubridate::epiyear(), "iso" uses lubridate::isoweek() / lubridate::isoyear().

Value

align_weeks() returns its input with an aligned date column added (data.frame method), or a tbl_now whose dates have been aligned and whose .delay has been recomputed.

week_2_date() returns the input data.frame with a new date column appended.

Note

Useful whenever week boundaries differ between systems or between years, which is the normal state of affairs for epiweek and ISO week data.

See Also

tbl_now(), whose align_weeks = TRUE argument does this at construction time; is_weekday(), which numbers weekdays the same way; complete_zeroes() for filling the weeks where nothing was reported; temporal_effects() for using week-of-year as a model term.

Examples

```
## ---- Plain data frames ------------------------------------------------

# Three dates falling on different weekdays.
df <- data.frame(date = as.Date(c("2022-11-02", "2022-11-07", "2022-11-13")))
weekdays(df$date)

# Snap them all back to the Sunday that starts their week.
aligned <- align_weeks(df, date_col = date)
aligned
weekdays(aligned$date_aligned)

# Or to Tuesday. Weekday numbers are ISO: 1 = Monday, so Tuesday is 2.
align_weeks(df, date_col = date, align_on_day = 2)

## ---- A tbl_now: making the delays whole numbers -------------------------

data(flusight)

# One state is enough to see the problem.
texas <- flusight[flusight$location_name == "Texas", ]
flutbl <- tbl_now(texas,
  event_date = "target_end_date",
  report_date = "as_of", case_count = "observation",
  strata = "location_name", verbose = FALSE
)

# `as_of` is sometimes a Saturday and sometimes a Wednesday, so some delays
# land between whole weeks.
mean(flutbl$.delay != round(flutbl$.delay))

# After aligning, every delay is a whole number of weeks.
flutbl <- align_weeks(flutbl)
mean(flutbl$.delay != round(flutbl$.delay))

## ---- Week numbers instead of dates --------------------------------------

# Data reported as "week 1 of 2024" and so on, with no usable date column.
df <- data.frame(
  epidemiological_week = 1:5,
  epidemiological_year = rep(2024, 5)
)

## week_2_date() turns those into the Sunday that starts each epiweek.
week_2_date(df,
  week_col = epidemiological_week,
  year_col = epidemiological_year
)
```

assign_tbl *Base R operations on a* tbl_now

Description

[Stable]

A tbl_now() is a tibble with extra attributes recording which column is the event date, which is the report date, and so on. These methods make sure those attributes survive ordinary base-R manipulation, so that x[1:10,], names(x) <- ... and x$new <- ... give you back a tbl_now rather than a bare data frame.

You never call them directly – they are what makes the operators work.

Usage

```
## S3 method for class 'tbl_now'
x[...]

## S3 method for class 'grouped_tbl_now'
x[...]

## S3 replacement method for class 'tbl_now'
names(x) <- value

## S3 replacement method for class 'grouped_tbl_now'
names(x) <- value

## S3 replacement method for class 'tbl_now'
x$name <- value

## S3 replacement method for class 'grouped_tbl_now'
x$name <- value
```

Arguments

x	A tbl_now object.
...	Passed to the underlying [method: rows and columns to keep.
value	For names<- and $<-, the replacement value.
name	For $<-, the column being assigned to.

Details

When an operation leaves the object unable to describe a nowcast – because it dropped or renamed the event-date column, say – the class cannot honestly be kept. In that case the result is **demoted** to a plain data frame (with a warning), rather than pretending to still be a tbl_now. Attributes that are still meaningful are preserved on the way down.

The same applies to dplyr verbs, through dplyr_reconstruct().

Value

A tbl_now object, or a plain data frame when the operation inrevised the class.

See Also

tbl_now() for the attributes being preserved; tbl_now_attributes() to check what survived; as_tibble() to drop the class on purpose; validate_tbl_now() to confirm the result is still well formed.

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat,
  event_date = onset_week, report_date = report_week,
  strata = gender, verbose = FALSE
)

# Subsetting rows keeps the class and everything it knows.
small <- dengue[1:10, ]
class(small)[1]
get_event_date(small)

# So does adding a column with `$<-`.
dengue$season <- ifelse(
  lubridate::month(dengue$onset_week) %in% 6:11, "wet", "dry"
)
class(dengue)[1]

# And renaming an unimportant column with `names<-`.
renamed <- dengue
names(renamed)[names(renamed) == "season"] <- "period"
get_event_date(renamed)

# But dropping the event date leaves nothing a nowcast could use, so the
# object is demoted to a plain tibble instead of lying about itself.
demoted <- suppressWarnings(dengue[, c("report_week", "gender")])
class(demoted)[1]
```

---

as_tbl_now *Transform an object into a* tbl_now

Description

[Stable]

Convert a supported object into a tbl_now. For a plain data.frame / data.table (or an existing tbl_now) you supply the event_date and report_date columns yourself. For objects produced by other packages the conversion is delegated to the matching tbl_now_from_*() converter, which already knows how to map that format – so those methods do **not** take event_date / report_date.

Usage

```
as_tbl_now(object, ...)

## S3 method for class 'tbl_now'
as_tbl_now(object, event_date, report_date, ...)

## S3 method for class 'data.frame'
as_tbl_now(object, event_date, report_date, ...)

## S3 method for class 'enw_preprocess_data'
as_tbl_now(object, ...)

## S3 method for class 'reporting_triangle'
as_tbl_now(object, ...)

## S3 method for class 'epidist_linelist_data'
as_tbl_now(object, ...)

## S3 method for class 'epidist_aggregate_data'
as_tbl_now(object, ...)

## S3 method for class 'tbl_ts'
as_tbl_now(object, report_date, event_date = NULL, ...)

## S3 method for class 'data.table'
as_tbl_now(object, event_date, report_date, ...)

## S3 method for class 'tbl_now_epinow2_snapshots'
as_tbl_now(object, ...)

## S3 method for class 'tbl_now_triangle_list'
as_tbl_now(object, ...)

## S3 method for class 'tbl_now_surveillance_list'
as_tbl_now(object, ...)
```

Arguments

`object`	An object to convert to a `tbl_now`.
`...`	Additional arguments forwarded to the relevant `tbl_now_from_*()` converter (and therefore to `tbl_now()`).
`event_date, report_date`	The event- and report-date columns, as tidy-select expressions – a bare column name or a string both work. Used for `data.frame`, `data.table`, `tbl_ts` (tsibble) and `tbl_now` inputs; for a tsibble, `event_date` defaults to the index. They are **not** arguments of the package-conversion methods (epinowcast, baselinenowcast, epidist), which carry their own date mapping.

Details

Package-specific inputs forward to a dedicated converter. **See that converter** for the extra arguments it accepts (e.g. `strata`, `max_delay`, `format`, `delays_unit`, ...), for which columns are carried over, and for the transformation notes / round-trip caveats in its *Round-trip* section:

- `enw_preprocess_data` (or a fitted `epinowcast` object) -> `tbl_now_from_epinowcast()`
- `reporting_triangle` (**baselinenowcast**) -> `tbl_now_from_baselinenowcast()`
- `epidist_linelist_data` / `epidist_aggregate_data` (**epidist**) -> `tbl_now_from_epidist()`
- `tbl_ts` (**tsibble**) -> `tbl_now_from_tsibble()`
- `data.table` -> `tbl_now_from_data_table()`

Anything passed through `...` is forwarded to the underlying converter (and on to `tbl_now()`), so options such as `event_units`, `now` or `verbose` can be supplied here too.

Value

A `tbl_now` object.

See Also

`tbl_now()` to build one from scratch; the converters this dispatches to – `tbl_now_from_epinowcast()`, `tbl_now_from_baselinenowcast()`, `tbl_now_from_epidist()`, `tbl_now_from_tsibble()`, `tbl_now_from_data_table` – and the `tbl_now_to_*()` functions that go the other way. The *One dataset, many nowcasts* article shows the round trip against each modelling package.

Examples

```
## For a plain data.frame this is a synonym for tbl_now(): you name the
# columns yourself.
data(denguedat)
as_tbl_now(denguedat, event_date = "onset_week", report_date = "report_week")

# For an object built by another nowcasting package you often do not name them,
# because that format already fixes which column is the event date and which
# is the report date. Here we send a tbl_now out to tsibble and bring it back.
if (requireNamespace("tsibble", quietly = TRUE)) {
  ndata <- tbl_now(denguedat,
    event_date = onset_week, report_date = report_week, verbose = FALSE
  )
  ts <- suppressWarnings(tbl_now_to_tsibble(ndata, verbose = FALSE))

  # Bare names and strings both work.
  as_tbl_now(ts, event_date = onset_week, report_date = report_week)
}
```

`as_tibble.tbl_now` *Coerce a* `tbl_now` *to a tibble or a data frame*

Description

[Stable]

`tibble::as_tibble()` and `as.data.frame()` methods for `tbl_now`. They drop the `tbl_now` class and metadata, returning a plain `tibble` / `data.frame`.

Set `compute_temporal_effects = TRUE` to **materialise the lazy** `temporal_effects()` **specification** with `compute_temporal_effects()` first, so the holiday / Fourier / calendar columns are present in the result. The input `tbl_now` is left unchanged (the spec is materialised on a copy).

The default is `FALSE` on purpose: **dplyr** uses these coercions internally as cheap declassers (for example inside `group_by()` and the data mask behind `mutate()` / `filter()` / `slice()`). Materialising there would both break the lazy `temporal_effects` contract and recurse, so materialising is strictly opt-in. To get a modelling-ready frame with the effects computed, either pass `compute_temporal_effects = TRUE` here or call `compute_temporal_effects()` on the `tbl_now` beforehand.

Usage

```
## S3 method for class 'tbl_now'
as_tibble(x, ..., compute_temporal_effects = FALSE)

## S3 method for class 'grouped_tbl_now'
as_tibble(x, ..., compute_temporal_effects = FALSE)

## S3 method for class 'tbl_now'
as.data.frame(x, ..., compute_temporal_effects = FALSE)

## S3 method for class 'grouped_tbl_now'
as.data.frame(x, ..., compute_temporal_effects = FALSE)
```

Arguments

`x`	A `tbl_now` (or `grouped_tbl_now`) object.
`...`	Passed on to the underlying `tibble::as_tibble()` / `base::as.data.frame()`.
`compute_temporal_effects`	Logical (default `FALSE`). When `TRUE`, materialise the lazy `temporal_effects()` spec before coercing so the temporal-effect columns are included.

Value

A `tibble` (for `as_tibble()`) or a `data.frame` (for `as.data.frame()`); it carries the temporal-effect columns when `compute_temporal_effects = TRUE`.

See Also

compute_temporal_effects(), temporal_effects()

Examples

```
data(denguedat)
df_now <- tbl_now(denguedat,
  event_date  = onset_week,
  report_date = report_week,
  t_effects   = temporal_effects(week_of_year = TRUE),
  verbose     = FALSE
)

## Plain coercion leaves the spec lazy (no temporal-effect columns):
".event_week_of_year" %in% names(tibble::as_tibble(df_now)) # FALSE

# Opt in to materialise the holiday / Fourier / calendar columns:
as_tibble(df_now, compute_temporal_effects = TRUE)
as.data.frame(df_now, compute_temporal_effects = TRUE)
```

---

as_tibble.tbl_nowcast *Coerce a* tbl_nowcast *into a* tibble

Description

[Stable]

Turns a fitted nowcast into an ordinary `tibble` you can plot, join or write out: one row per event date and quantile level by default, or one row per posterior draw with `type = "draws"`.

Not every engine keeps draws. When the backend returned only summarised quantiles, `type = "draws"` has nothing to give you.

Usage

```
## S3 method for class 'tbl_nowcast'
as_tibble(x, ..., type = c("quantiles", "draws"))
```

Arguments

x	A tbl_nowcast.
...	Unused.
type	Either "quantiles" (default) for the tidy quantile predictions, or "draws" for the posterior draws.

Details

Registered in `.onLoad()` rather than assigned with `S7::method(as_tibble, tbl_nowcast) <-`, which is what the neighbouring `print()` and `as.data.frame()` methods use. Assigning an S7 method onto an *imported* generic copies that generic into this namespace, and `tibble::as_tibble()` carries `rownames = pkgconfig::get_config(...)` as a default argument – so the copy makes `R CMD check` report **pkgconfig** as an undeclared `::` import of a package this one never uses. Plain S3 registration dispatches on `class(x)`, which for an S7 object is `"tbl.now::tbl_nowcast"`, and copies nothing.

The registered class string cannot be spelled as a method name – there is no writable `as_tibble.tbl.now::tbl_nowcast` – but the function itself is still *named* for the method it implements, so that `R CMD check` can resolve this topic's usage section back to an object that exists.

Value

A `tibble`. For `type = "quantiles"`, one row per event date, stratum and quantile level, with the quantile level in `.quantile_level` and the predicted count in `.value`. For `type = "draws"`, one row per draw.

See Also

tidy() for the same content with the engine's metadata attached; autoplot() to plot it; `score_nowcast()` to score it against observed counts.

Examples

```
predictions <- data.frame(
  onset_week = as.Date("2020-01-05"),
  .quantile_level = c(0.5, 0.9), .value = c(10, 14)
)
nc <- tbl_nowcast(predictions = predictions, method = "toy", event_date = "onset_week")
tibble::as_tibble(nc)
```

`autoplot.tbl_now` *Diagnostic* `autoplot` *for a* `tbl_now`

Description

[Experimental]

Produces a multi-panel diagnostic overview of a `tbl_now` using `ggplot2::ggplot()` and **patchwork**. The gallery is a **matrix with one column per process**: the **case counts** on the left, the **reporting delay** next to them, and — when the object declares a revision axis — the **revision process** on the right. Each row asks the same question of every process, so an object with two dates comes out two columns wide and one with three dates three columns wide. You choose which panels to draw with the `panels` argument, and the number of columns follows: a selection covering one family only (`panels = "calendar"`) is one column.

A row a process cannot answer — weekly data has no day-of-week panel — leaves that cell empty rather than closing the gap, so the columns keep their meaning all the way down.

Case-count panels

- `"delay_distribution"` — a (case-count weighted) histogram of the reporting delay (`.delay`). For `count-cumulative` data this panel instead shows the *cumulative growth by delay*: boxplots (on a log scale, with a dashed reference at 1) of the ratio of each event date's cumulative count at a delay to its cumulative count at the previous delay. A ratio above 1 is an upward revision, below 1 a downward one, and the boxes converge to 1 as reporting completes.
- `"epidemic"` — the latest reported case counts per `event_date`, with a dashed vertical line marking where the data become incomplete (less than `level` of the delay distribution has arrived). Holidays from the attached `temporal_effects()` spec are marked with dots.
- `"calendar_weekday"`, `"calendar_week"`, `"calendar_month"` — boxplots of the *normalized* case effect (each event date's cases divided by the overall mean, so 1 is average) by day of week, epidemiological week, or month.
- `"calendar_holiday"` — the same normalized boxplots by **day type**, titled "Weekend and/or holiday effects" because that is what the day types are. The categories follow the attached `temporal_effects()` spec: a holiday calendar and a `weekend` effect together give `Weekday` / `Weekend` / `Holiday`, a calendar alone gives `Non-holiday` / `Holiday`, and a `weekend` effect alone gives `Weekday` / `Weekend`. A holiday falling on a weekend counts as a holiday.
- `"calendar_holiday_lag"` — the same normalized boxplots by **position relative to the nearest holiday**, as asked for by `holiday_lags` (see `temporal_effects()`): `"2 before"`, `"1 before"`, `"Holiday"`, `"1 after"`, ..., plus `"Other"` for every other day as the reference. It shows exactly the days the `..._holiday_lag_k` / `..._holiday_lead_k` columns flag — weekends and other holidays are skipped when counting working days — so you can see whether the lags you asked for are the ones that matter.
- `"seasonality"` — a **cycles** periodogram of the incidence series whose dominant peak suggests a Fourier season length for `temporal_effects()`.

Reporting-delay panels (to inspect *delay effects*)

- `"delay_weekday"`, `"delay_week"`, `"delay_month"` — boxplots of the *normalized* mean reporting delay (each event date's mean delay divided by the overall mean delay, so 1 is average) by day of week, epidemiological week, or month; these reveal whether the delay itself has a calendar pattern. Normalizing keeps them on the same scale as the case-count calendar panels and makes them comparable across strata.
- `"delay_holiday"`, `"delay_holiday_lag"` — the reporting-delay twins of the two holiday panels above: the *normalized* mean delay by day type and by position relative to the nearest holiday. These are often the more telling pair — a holiday usually does not change how many cases occur, but it very much changes how long they take to be reported.
- `"delay_seasonality"` — a **cycles** periodogram of the mean-delay series, whose peak marks a cycle in the reporting delay (e.g. a weekly reporting rhythm).

Revision-process panels (only when the object declares a revision axis)

- `"revision_distribution"` — the reporting-delay histogram's twin on the revision axis: a (case-count weighted) histogram of `.revision_delay`, the time from a report to its resolution. A case still `"pending"` has no resolution and so no revision delay, and does not appear.

- `"revision_weekday"`, `"revision_week"`, `"revision_month"`, `"revision_holiday"`, `"revision_holiday_lag"`, `"revision_seasonality"` — the same calendar and periodogram questions asked of the dates resolutions arrived on.

Every panel is colour-coded by the process it describes — **red** for the reporting-delay panels, **green** for the case-count (epidemic) ones — and says which one it is in its subtitle, so a single panel still reads on its own.

Which panels are available depends on the object. The **calendar/delay** panels follow the event unit: daily data offers day-of-week **and** week-of-year panels, weekly data week-of-year, monthly data month-of-year. The four **holiday** panels describe the attached `temporal_effects()` spec, so they appear only when there is one to describe: `"calendar_holiday"` / `"delay_holiday"` need a `holidays` calendar or a `weekend` effect, and the two lag panels additionally need a non-zero `holiday_lags`. Requesting a holiday panel without the matching effect warns and skips it. The spec is read directly, so you do **not** need to call `compute_temporal_effects()` first.

The delay panels are computed on the *complete* portion of the series (event dates on or before the incompleteness line) so the recent reporting truncation does not bias them.

Usage

```
## S3 method for class 'tbl_now'
autoplot(
  object,
  ...,
  panels = "all",
  by_strata = FALSE,
  strata = NULL,
  measure = c("percent", "normalized"),
  by_revision_type = FALSE,
  level = 0.95,
  plotly = FALSE,
  size = 1,
  linewidth = 1,
  palette = .tbl_now_palette(),
  delay_distribution_xlim = NULL,
  event_date_xlim = NULL,
  calendar_effect_xlim = NULL,
  seasonality_xlim = NULL
)
```

Arguments

`object`	A `tbl_now` object.
`...`	Unused; present for compatibility with `ggplot2::autoplot()`.
`panels`	Which panels to draw. Either a vector of the concrete keys listed above, or one of the aliases `"all"` (default; every applicable panel), `"calendar"` (the case-count calendar panels) or `"delay_calendar"` (the reporting-delay calendar panels). Selecting a single panel returns that panel as a plain **ggplot2** object instead of a **patchwork**.

by_strata
: Logical (default FALSE). When TRUE, every panel is split by stratum: the calendar / delay boxplots become dodged boxes (one per stratum, side by side), the epidemic process and both periodograms become one coloured line per stratum (no area fill), and the delay distribution becomes dodged bars (one per stratum). The boxplots are then normalized **per stratum** (1 = that stratum's own average) so the calendar pattern is comparable across strata. Colours use a **viridis** scale. In this mode the holiday dots are omitted from the epidemic panel.

strata
: Character vector of column names to group by when by_strata = TRUE. NULL (default) uses the object's strata (see get_strata()); pass a subset (e.g. strata = "gender") to group by only some of them. Ignored when by_strata = FALSE.

measure
: How to express the calendar-effect boxplots (the day-of-week, week-of-year and month-of-year panels; every other panel ignores it, and the two **holiday** pairs are always "normalized" — see below).

  - "normalized" — the value divided by its overall mean, so 1 (the dashed line) marks an average level. Case-count panels normalize the cases per event date; delay panels normalize the mean reporting delay.
  - "percent" (default) — the **share of cases** falling in each group, as a percentage, so the box reads directly as "10% of cases at the weekend versus 90% on weekdays" with the IQR around it. One observation per calendar block: the seven weekdays (and the day types) are shared out within each **week**, the holiday lags within each **month**, and the epidemiological weeks and months within each **year**. The reporting-delay panels then switch from the event date to the **report date**, so they answer "what share of the reports *arrive* on a weekend?". Needs Date event/report columns.

  The four holiday panels ("calendar_holiday", "calendar_holiday_lag" and their delay twins) ignore measure and are always drawn "normalized". Their categories are not equal-sized parts of a calendar block — the weekend is two days in seven — so a share would mostly report how the calendar is built rather than how the data behave: "29% of cases at the weekend" is average, not low.

by_revision_type
: Logical (default FALSE). When TRUE, the two delay-distribution panels are split by how each case eventually resolved: confirmed, pending, retracted and unknown, stacked, in the palette's outcome colours (see tbl_now_palette()). Ignored on an object with no revision axis, and when by_strata = TRUE, which already spends the fill on the strata. It defaults to FALSE here because the gallery is read as a grid of shapes, and to TRUE in plot_delay_distribution(), where the panel is the whole plot.

level
: Completeness level used for the incompleteness line in the "epidemic" panel (and to trim the delay panels). The line is drawn at now - q, where q is the level quantile of the delay distribution. With the default 0.95, the line marks where at least 5 percent of delays are yet to arrive.

plotly
: If TRUE, return an interactive **plotly** widget (the panels stacked) instead of a static **patchwork**. Default FALSE.

size
: Multiplier on every point, outlier and annotation-label size the panels draw. Default 1. It multiplies rather than replaces, so a panel that deliberately draws one mark larger than another keeps that difference at any setting.

`linewidth` Multiplier on every line, boxplot outline and reference-line width the panels draw. Default 1.

`palette` A named colour palette (see `tbl_now_palette()`). Every colour is named for the role it plays, so overriding one role re-themes every panel that uses it.

`delay_distribution_xlim`, `event_date_xlim`, `calendar_effect_xlim`, `seasonality_xlim`
Optional length-2 vectors giving the x-axis limits for the corresponding panel (delay-distribution histogram, epidemic process, calendar-effect boxplots, incidence periodogram). `NULL` (default) lets each panel pick its own range. For `event_date_xlim` pass `Date`s; the others take numeric limits.

Value

A **patchwork** object combining the selected panels, one column per process, or — when a single panel is selected — that panel as a **ggplot2** object.

See Also

`diagnostic_plot()` for the companion gallery, which looks at the *reporting process* – when reports arrived and whether any of it is artificial – rather than at the case counts; the panels here as standalone functions: `plot_observed_cases()`, `plot_delay_distribution()`, `plot_cycles()`, calendar_effect_plots and `plot_delay_drift()`; summary() and `diagnose()` for the same information as tables. The *Diagnosing a tbl_now* article reads the panels one at a time.

Examples

```
data(denguedat)
# A few recent months keep the example fast; the panels look the same on
# twenty years of data, they just take longer to draw.
recent <- denguedat[denguedat$onset_week >= as.Date("2010-11-01"), ]
dengue <- tbl_now(recent,
  event_date = "onset_week",
  report_date = "report_week", strata = "gender", verbose = FALSE
)
autoplot(dengue)

# A single panel comes back as a plain ggplot, so it can be selected, split
# by stratum and then treated like any other plot. `panels` takes any of the
# names listed above; `by_strata = TRUE` works on the whole gallery too, but
# one panel keeps the example quick.
autoplot(dengue, panels = "delay_week", by_strata = TRUE)

# Panel-specific arguments compose with it: zoom the delay panel to 0-10 weeks
autoplot(dengue, panels = "delay_distribution", delay_distribution_xlim = c(0, 10))
```

`autoplot.tbl_nowcast` *Plot a nowcast*

Description

[Experimental]

Draws a fan chart of a tbl_nowcast: the counts reported so far as grey columns, one shaded band per central prediction interval over them, and the median as a line, so that the size of the correction the model is applying is visible as the gap between the bars and the fan.

Usage

```
## S3 method for class 'tbl_nowcast'
autoplot(object, ..., levels = NULL,
  show_reported = TRUE, colour = NULL, linewidth = 1, date_lim = NULL,
  ylim = NULL, palette = .tbl_now_palette())
```

Arguments

`object`	A tbl_nowcast object.
`...`	Unused; present for compatibility with `ggplot2::autoplot()`.
`levels`	Numeric vector of central interval widths to shade. Defaults to the widest intervals available in the object.
`show_reported`	Logical. Whether to draw the cases **reported so far** as columns under the fan – `get_latest_reported_cases()` on the object's own source data, so the bars are what the model was actually shown as of `now`, not what those dates eventually reached. The vertical gap between the top of a bar and the fan is the correction the nowcast is making, which is the whole reason to draw it. Requires the nowcast to carry its source data. The bars are one period wide, taken from `get_event_units()`. A fixed width would draw hairlines on a weekly series, and one wider than the step would make **ggplot2** stack overlapping bars, so each would show several periods' counts rather than its own.
`colour`	Colour of the fan. Defaults to the `tbl.now` palette's green: a nowcast is an estimate of the **epidemic** process (cases by event date), which the package always draws in green, with red reserved for the reporting process.
`linewidth`	Multiplier on the width of the median line. Default `1`; it multiplies rather than replaces the geom's own width.
`date_lim`	Length-2 vector of event-axis limits, as `Dates` (or as numbers on a numeric event axis). `NA` in either position leaves that end alone. A nowcast covers the whole series but only *corrects* its final periods, so the interesting part is usually the last few weeks; this zooms on to them. The limits are applied with `ggplot2::coord_cartesian()`, so they **crop** the drawn plot rather than filter the data. That matters here: a scale limit would drop the out-of-range rows before the ribbon is built, which cuts the fan off at the boundary instead of letting it run to the edge.

ylim	Length-2 vector of count-axis limits, applied the same way. NULL (default) leaves the axis to **ggplot2**. Note that a stratified nowcast facets with scales = "free_y", so one pair of limits is imposed on every panel.
palette	A named colour palette (see tbl_now_palette()). Each colour is named for the role it plays, so overriding one role re-themes the plot.

Details

A named function registered in .onLoad(), like the neighbouring tidy() and as_tibble() methods and for the same two reasons.
S7::method(autoplot, tbl_nowcast) <- would be shorter, but method<- is a replacement function, so R rewrites the call as an assignment back to autoplot and leaves a **copy of ggplot2's generic in this namespace**. That copy is what R CMD check sees when it decides autoplot is a generic this package owns and exports, which in turn makes every autoplot.* function here look like an S3 method that was never registered. Plain registration copies nothing.

The function is also *named* for the method it implements, rather than being assigned anonymously into the generic, so that R CMD check can resolve this topic's usage section back to an object that exists.

Value

A ggplot object.

See Also

run_nowcast(), nowcast_ensemble()

Examples

```
predictions <- tidyr::expand_grid(
  onset_week = as.Date("2020-01-05") + seq(0, 28, by = 7),
  .quantile_level = c(0.05, 0.25, 0.5, 0.75, 0.95)
)
predictions$.value <- 10 + 30 * predictions$.quantile_level
nc <- tbl_nowcast(predictions = predictions, method = "toy", event_date = "onset_week")

autoplot(nc)

# Zoom on to the corrected weeks without dropping the rows that build the fan.
autoplot(nc, date_lim = c(as.Date("2020-01-19"), as.Date("2020-02-02")))
```

---

calendar_effect_plots *Calendar effects on the case counts or on the reporting delay*

Description

[Stable]

One panel of `autoplot()`, drawn on its own. Each function shows the same boxplots the corresponding `autoplot()` panel does, for one calendar grouping:

- `plot_day_of_week_effects()` — by day of week (daily data only).
- `plot_week_of_year_effects()` — by epidemiological week.
- `plot_month_of_year_effects()` — by month (monthly data only).
- `plot_holiday_effects()` — by **day type** (`Weekday` / `Weekend` / `Holiday`, following the attached `temporal_effects()` spec).
- `plot_weekend_effects()` — the same panel, on an object that may not carry a spec yet: it attaches `temporal_effects(weekend = TRUE)` when there is no weekend effect already, so the weekend boxes appear without a separate `add_temporal_effects()` call. The spec goes on a copy — your object is not modified — and a calendar already attached still contributes its `Holiday` box. Daily data only, since a weekend is a property of the day.
- `plot_holiday_lag_effects()` — by position relative to the nearest holiday (`"1 before"`, `"Holiday"`, `"1 after"`, ..., plus `"Other"`).

`type` picks which process to describe: `"epidemic"` (green — how the *cases* vary by calendar group), `"report"` (red — how the *reporting* does), or `"revision"` (ochre — how resolved cases arrive on revision dates).

The three day-type / holiday-lag functions have no `measure` argument: they are always normalized. Their categories are not equal-sized parts of a calendar block — the weekend is two days in seven — so a percentage share would mostly restate the calendar rather than the data ("29% of the cases at the weekend" is average, not low). The day-of-week, week-of-year and month-of-year functions keep both measures.

Use these when you want one effect, in its own figure, at its own size; use `autoplot()` when you want the diagnostic grid in one call. Everything else is the same: `autoplot(x, panels = "calendar_weekday")` and `plot_day_of_week_effects(x)` return the identical plot.

Usage

```
plot_day_of_week_effects(
  x,
  type = c("epidemic", "report", "revision"),
  measure = c("percent", "normalized"),
  ...
)

plot_week_of_year_effects(
  x,
  type = c("epidemic", "report", "revision"),
  measure = c("percent", "normalized"),
  ...
)
```

```
plot_month_of_year_effects(
  x,
  type = c("epidemic", "report", "revision"),
  measure = c("percent", "normalized"),
  ...
)

plot_holiday_effects(x, type = c("epidemic", "report", "revision"), ...)

plot_weekend_effects(
  x,
  type = c("epidemic", "report", "revision"),
  weekend_days = c("Sat", "Sun"),
  ...
)

plot_holiday_lag_effects(x, type = c("epidemic", "report", "revision"), ...)
```

Arguments

x	A tbl_now() object.
type	"epidemic" (default) for the case-count effect, "report" for the reporting-delay one, or "revision" for revision-date arrivals.
measure	"percent" (default) for the share of cases in each group — "10% of cases in week 1 versus 3% in week 2" — with the IQR around it, or "normalized" for the value divided by its overall mean (1 = average). See autoplot.tbl_now() for the blocks the percentages are taken over. The day-type and holiday-lag functions do not take it; they are always normalized.
...	Further arguments passed to autoplot.tbl_now(), e.g. by_strata, strata, plotly or palette.
weekend_days	Character vector naming the weekend days (default c("Sat", "Sun")), as in is_weekday(). Used **only** when plot_weekend_effects() has to attach the effect itself; an object that already carries a weekend effect keeps the definition it was given.

Value

A **ggplot2** object (or a **plotly** widget when plotly = TRUE).

See Also

autoplot.tbl_now(), plot_cycles(), plot_delay_distribution(), plot_observed_cases(); temporal_effects() and add_temporal_effects() for the specification the day-type and holiday-lag panels describe.

Examples

```
data(denguedat)
```

```
# First few years only, to keep the example quick; the full data works the same.
dengue_now <- tbl_now(denguedat[1:2500, ], onset_week, report_week, verbose = FALSE)

# How the cases vary by epidemiological week
plot_week_of_year_effects(dengue_now)

# The weekend on its own, on daily data, with no spec to attach first
days <- seq(as.Date("2021-01-01"), as.Date("2021-06-30"), by = "day")
daily_now <- tbl_now(
  data.frame(event_date = days, report_date = days + 1),
  event_date, report_date, verbose = FALSE
)
plot_weekend_effects(daily_now)

# By month, on monthly-unit data. `type` picks the process and `measure`
# picks the scale; both compose, and work the same way on every calendar
# function here. (The day-type and holiday-lag panels are always
# normalized, so they take `type` but not `measure`.)
monthly_now <- tbl_now(
  data.frame(
    event_date  = seq(as.Date("2018-01-01"), as.Date("2021-12-01"), by = "month"),
    report_date = seq(as.Date("2018-02-01"), as.Date("2022-01-01"), by = "month")
  ),
  event_date, report_date,
  event_units = "months", report_units = "months", verbose = FALSE
)
plot_month_of_year_effects(monthly_now, type = "report", measure = "normalized")

if (requireNamespace("almanac", quietly = TRUE)){

  ## By day type (weekday / weekend / holiday), once a holiday calendar is attached
  holiday_now <- dengue_now |>
  add_temporal_effects(temporal_effects(weekend = TRUE, holidays = almanac::cal_us_federal()))
  plot_holiday_effects(holiday_now)

  # By position relative to the nearest holiday
  holiday_lag_now <- dengue_now |>
   add_temporal_effects(temporal_effects(holidays = almanac::cal_us_federal(), holiday_lags = 2))
  plot_holiday_lag_effects(holiday_lag_now)

}
```

`censoring` *Record a report or a delay as a bound rather than a fact*

Description

[Stable]

Surveillance data is full of dates that are not really dates. A case with onset in March turns up in December; a report date is missing altogether; a system codes "never reported" as `2222-02-22`. Deleting those records throws away real cases, and believing them drags the estimated delay distribution to the right until the nowcast thinks reporting is far slower than it is.

These functions **censor** instead: the object keeps the case, but records its delay as *"at least this long"* rather than *"exactly this long"*.

There are two axes to censor and three ways to say which rows, so there are six verbs. On the **reporting** axis (event date to report date, the `is_censored_report` flag):

- `censor_reports()` – rows matching a condition get a **replacement report date** (the missing ones become `now`, say) and the flag.
- `censor_reporting_delays()` – the same, said as a **delay** rather than a date; with no replacement it only sets the flag.
- `censor_reporting_delays_above()` – **considers as censored every delay longer than** `max_delay`, in the object's event units, and leaves every other row alone. This is the one you want when the threshold *is* the rule: "anything that took more than 60 days is a lower bound, not a measurement".

On the **revision** axis (report date to resolution, the `is_censored_revision` flag), the same three:

- `censor_revisions()` – rows matching a condition get a **replacement revision date** and the flag.
- `censor_revision_delays()` – the same, said as a delay from the report.
- `censor_revision_delays_above()` – **considers as censored every revision delay longer than** `max_delay`, in the object's revision units: a laboratory result that took months is a case you have stopped believing the turnaround of.

Usage

```
censor_reporting_delays_above(x, max_delay, verbose = TRUE)

censor_reports(x, condition, to_report = get_now(x), verbose = TRUE)

censor_reporting_delays(x, condition, to_delay = NULL, verbose = TRUE)

censor_revisions(x, condition, to_revision = get_now(x), verbose = TRUE)

censor_revision_delays(x, condition, to_delay = NULL, verbose = TRUE)

censor_revision_delays_above(x, max_delay, verbose = TRUE)
```

Arguments

`x`	A `tbl_now` object. The three revision verbs require one that carries a revision process (see add_revision_date()).
`max_delay`	Numeric. Every delay **strictly greater** than this is considered censored; the rest are left alone. In the object's event units for `censor_reporting_delays_above()`, revision units for `censor_revision_delays_above()`.

verbose — Logical. Whether to report how many rows were affected. Default TRUE.

condition — An unquoted expression evaluated in x, as in dplyr::filter(). Rows where it is TRUE are censored.

to_report — The replacement report date for the matching rows: a single value, or one per row of x. Must match the class of the report column (a Date, or a number for a numeric axis). Defaults to get_now(x) – the case has not been reported as of now, which is the whole point of the censoring flag. NULL leaves the dates alone and only sets the flag.

to_delay — The replacement delay for the matching rows. For censor_reporting_delays() it is in the object's **event** units and the report date becomes event_date + to_delay; for censor_revision_delays() it is in **revision** units and the revision date becomes report_date + to_delay, because that is what .revision_delay measures. A single number or one per row. NULL (the default) leaves the dates alone and only sets the flag. It must be a **whole number** of those units, on every axis: there is no such date as half a day later, and a calendar axis used to bend 2.5 to 2 and 3.5 to 4 without saying so.

to_revision — The replacement revision date for the matching rows: a single value, or one per row of x. Must match the class of the revision column. Defaults to get_now(x) – the case has not been resolved as of now. NULL leaves the dates alone and only sets the flag. Pending cases are skipped; see *Pending cases are skipped*.

Details

The reporting delay is read from the generated .delay column (report date minus event date, in the object's event units); the revision delay from .revision_delay (revision date minus report date, in revision units). Existing censoring flags are merged rather than overwritten, so a delay that was already censored stays censored, and the flag column is created (as .is_censored_report / .is_censored_revision) when the object has none.

The threshold functions keep the case **and its date**. Nothing is deleted and no outcome is rewritten: the flag says the delay is a bound rather than a measurement, and it is up to the model to use that. A case that was confirmed after 200 days is still a confirmed case, and get_latest_revised_cases() still counts it.

condition is evaluated inside the data, like a dplyr::filter() expression, so it can name any column – including the generated .delay. Rows where it comes out NA are **not** censored: a condition that cannot be evaluated is not a condition that was met.

The four verbs that take a condition *can* move a date, and then they rebuild the object: the generated numeric and delay columns are recomputed, and now moves **forward** when a replacement lands after it, never backwards – now is where you are standing, not the last date in the data. Nothing stops a replacement from landing before the date it is measured from; that is a negative delay, and validate_tbl_now() says so.

Any temporal-effect column materialised **on the report date** (.report_*, from compute_temporal_effects()) is dropped when the report date moves, because it describes a date that has just changed; run compute_temporal_effects() again to rebuild it. The .event_* ones are kept – the event dates never move – and neither is touched by the revision verbs.

Value

A `tbl_now` with that axis's censoring column updated, creating it when absent (`.is_censored_report` or `.is_censored_revision`), and with the dates replaced where a replacement was asked for. The three reporting verbs touch `is_censored_report` and the report date; the three revision verbs touch `is_censored_revision` and the revision date. Neither rewrites `revision_type`, and nothing is ever deleted.

Pending cases are skipped

`"pending"` means **reported and still waiting**, so a pending case has no revision date – that is the whole difference between it and a resolution that was never recorded. Writing a date onto one would assert a resolution that never happened, and make the case look resolved to everything counting arrivals on the revision axis.

So `censor_revisions()` and `censor_revision_delays()` **skip pending rows** when they would write a date, and say how many they skipped. To censor a case that really was resolved but whose date is missing, make sure its `revision_type` says so first. Flagging without a replacement is not affected: no date is written, so there is nothing to contradict.

See Also

add_is_censored_report() and change_is_censored_report() to set the flag by hand, and add_is_censored_revision() for the revision axis; `diagnose_revision_delay()` and `plot_delay_distribution()` to find the threshold worth using; `diagnose_truncation()` for the delays that are missing rather than long; `complete_zeroes()` for the opposite problem.

Examples

```
# Four cases, one of which took 300 days to be reported.
df <- data.frame(
  onset = as.Date("2020-01-01") + c(0, 0, 1, 2),
  reported = as.Date("2020-01-01") + c(1, 5, 2, 300)
)
tn <- tbl_now(df,
  event_date = onset, report_date = reported,
  data_type = "linelist", verbose = FALSE
)
tn$.delay

# Anything slower than 60 days is recorded as a lower bound, not a fact.
censored <- censor_reporting_delays_above(tn, max_delay = 60)
censored[[get_is_censored_report(censored)]]

# The same rule written by hand, and capped at 60 days as well, so the
# 300-day outlier stops dominating the delay distribution.
capped <- censor_reporting_delays(tn, .delay > 60, to_delay = 60, verbose = FALSE)
capped$.delay

## ---- Reports that never arrived ---------------------------------------

# A missing report date, and a system that codes "never" as a date in 2222.
```

```
messy <- data.frame(
  onset = as.Date("2020-01-01") + 0:3,
  reported = as.Date(c("2020-01-03", NA, "2222-02-22", "2020-01-06"))
)
messy_now <- suppressWarnings(tbl_now(messy,
  event_date = onset, report_date = reported,
  data_type = "linelist", units = "days", verbose = FALSE,
  now = as.Date("2020-01-10")
))

# Both are "not reported yet", so both become `now` and are flagged censored.
# Wrapped in suppressWarnings because the object keeps warning about the
# dates being fixed.
fixed <- suppressWarnings(censor_reports(messy_now,
  is.na(reported) | reported > as.Date("2100-01-01"),
  verbose = FALSE
))
fixed[[get_report_date(fixed)]]
fixed[[get_is_censored_report(fixed)]]

## ---- The revision counterpart ----------------------------------------

cases <- data.frame(
  onset = as.Date("2021-01-04") + 0:4,
  visit = as.Date("2021-01-05") + 0:4,
  result = as.Date("2021-01-05") + 0:4 + c(1, 2, 1, 90, 2),
  outcome = rep("confirmed", 5)
)
flu <- tbl_now(cases,
  event_date = onset, report_date = visit,
  revision_date = result, revision_type = outcome,
  data_type = "linelist", verbose = FALSE
)

# That one is flagged; all five stay confirmed, and the date is kept.
flagged <- censor_revision_delays_above(flu, 30, verbose = FALSE)
flagged[[get_is_censored_revision(flagged)]]
table(flagged[["outcome"]])

# The condition form: cap that turnaround at 30 days from the report, which
# moves the revision date to match.
capped_lab <- censor_revision_delays(flu, .revision_delay > 30,
  to_delay = 30, verbose = FALSE
)
capped_lab$.revision_delay

## A pending case has no resolution date, so there is nothing to censor --
# it is skipped rather than given a date it never had.
waiting <- flu
waiting[["outcome"]][2] <- "pending"
waiting[["result"]][2] <- as.Date(NA)
waiting <- change_revision_date(waiting, "result", "outcome")
out <- censor_revisions(waiting, is.na(result), verbose = FALSE)
```

```
out[["result"]][2] # still NA
```

`complete_zeroes` *Fill in the days when nothing was reported*

Description

[Stable]

Surveillance data records what happened, not what didn't. If no dengue case with onset on 3 January was reported on 5 January, there is simply no row for that combination – which is *not* the same as a row saying zero, even though it means the same thing.

Most nowcasting models need the difference spelled out. They work on a complete rectangle of (event date x report date) cells, and a missing cell is ambiguous: it could be a genuine zero, or a delay so long the report has not arrived yet. `complete_zeroes()` writes the genuine zeros in explicitly, for every stratum, leaving only the not-yet-reported cells absent.

Usage

```
complete_zeroes(x, max_delay = NULL, until = NULL)
```

Arguments

`x`	A `tbl_now` object.
`max_delay`	Maximum delay to fill. For example if set to 5 it will complete with 0's all reports with delays 0 to 4. But will not fill other delays (say 6)
`until`	Event date to complete up to. `NULL` (the default) completes to whichever is later, the object's `get_now()` or the last event date present in the data. Completing only up to the last *observed* event date would leave a gap precisely at the `now` edge, because an event date with no reports at all does not appear in the data; several downstream converters build their time grid from the rows they are given and would silently stop short. A supplied `until` is never allowed to truncate below the data, and has no effect beyond the `now`: an event date later than the `now` cannot carry any report on or before it, so no row would survive for it.

Temporal effects:
If `x` arrived with materialised `temporal_effects()` columns, they are **recomputed on the completed grid** before the result is returned, so the rows this function adds carry their own calendar effects rather than `NA`. A lazy specification that has not been computed yet stays lazy.

Details

Zeros are only filled where a report *could* have arrived: cells with a report date on or before the event date's `now`, and within `max_delay`. Filling beyond that would invent observations from the future.

Rows with a missing date:

A row whose event or report date is NA has no cell on the rectangle, so it takes no part in the grid: the bounds (max_delay, the first and last event date, the last report date) are all computed ignoring it. It is still a case, though, so it is **carried through unchanged** rather than dropped – use censor_reports() to give it a bound, or dplyr::filter() to remove it, if you would rather it were on the grid or gone. Only an object in which *every* row is missing one of the two dates is refused, because then there is no grid to complete at all.

Value

A tbl_now object with the same columns as x, plus the rows that were implicitly zero, carrying 0 in the case_count column. Explicit missing counts in the input remain NA; only cells created by complete_zeroes() are filled. The data type is preserved, as are any computed temporal-effect columns (recomputed over the new rows).

See Also

to_count() for the data shapes this operates on; censor_reporting_delays_above() for the opposite problem, delays that are too long; diagnose_missing() and diagnose_truncation() to find the gaps first; plot_reporting_triangle() to see the rectangle being filled.

Examples

```
ndata <- dplyr::tibble(
  event = rep(c(
    as.Date("2020/01/01"), as.Date("2020/01/01"),
    as.Date("2020/01/02"), as.Date("2020/01/04"),
    as.Date("2020/01/04")
  ), 2),
  report = rep(c(
    as.Date("2020/01/01"), as.Date("2020/01/02"),
    as.Date("2020/01/02"), as.Date("2020/01/04"),
    as.Date("2020/01/05")
  ), 2),
  n = rpois(10, lambda = 5),
  sex = c(rep("Male", 5), rep("Female", 5))
)
ndata <- tbl_now(ndata,
  event_date = event, report_date = report,
  verbose = FALSE, strata = sex, case_count = n, data_type = "count-incidence"
)

# Nothing happened on 2020-01-03, so the data has no row for it at all.
sort(unique(ndata$event))

## complete_zeroes() writes that absence down as an explicit zero, for every
# stratum, so a model can tell "no cases" from "not reported yet".
filled <- complete_zeroes(ndata)
sort(unique(filled$event))
nrow(ndata)
nrow(filled)
```

```
# Also works for count-cumulative
ndata |>
  to_count("count-cumulative") |>
  complete_zeroes() |>
  dplyr::arrange(event, sex, report)
```

`covid_colombia` *COVID-19 Notifications – Colombia 2020-2023*

Description

Daily case counts of COVID-19 from Colombia's national epidemiological surveillance system (INS), aggregated by notification date, diagnosis date, and sex. Each row is a unique combination of these three variables together with the number of cases `n`.

Usage

```
covid_colombia
```

Format

A data frame with 35,501 rows and 4 variables:

notification_date `Date`. The event date – when the case was notified / symptom onset was recorded.

diagnosis_date `Date`. The report date – when the laboratory diagnosis was registered in the national system.

sex `character`. Biological sex of the case: `"Female"` or `"Male"`.

n `integer`. Number of cases with this (notification_date, diagnosis_date, sex) combination.

Details

In the nowcasting context the **event date** is `notification_date` (when the case symptom onset was recorded in the system) and the **report date** is `diagnosis_date` (when the laboratory result was entered). The delay between the two reflects the time from symptom onset to laboratory revision and data entry – the quantity the nowcasting model estimates and corrects for.

The dataset covers the full first three years of the Colombian epidemic (2020-03-02 to 2023-03-03) and was used in the NobBS comparison study to benchmark the diseasenowcasting / diseasenowcast2 engine.

Source

Instituto Nacional de Salud (INS), Colombia. Data accessed via the SIVIGILA open-data platform and pre-processed for the diseasenowcasting benchmarking study.

See Also

`tbl_now()` to declare the date columns; summary() and `diagnose()` to inspect the result; the package's other datasets – denguedat, mpoxdat, flusight, covid_colombia, covid_us and hai_bucaramanga.

Examples

```
data(covid_colombia)

# A stratified tbl_now: event = notification, report = diagnosis.
tn <- tbl_now(
  covid_colombia,
  event_date  = notification_date,
  report_date = diagnosis_date,
  strata      = sex,
  case_count  = n,
  data_type   = "count-incidence",
  verbose     = FALSE
)
tn
```

`covid_us` *covid_us: CDC COVID-19 Case Surveillance Public Use Data (2020)*

Description

A compact aggregation of the U.S. CDC's individual-level COVID-19 case surveillance database. It is the package's worked example for two different things: **batch reporting**, and the **revision process** – the optional third date a surveillance record can carry.

Usage

```
data(covid_us)
```

Format

A data frame with 192,953 rows and six variables:

onset_dt `Date`. The event date – symptom onset.

pos_spec_dt `Date`. The report date – collection of the first positive specimen.

cdc_report_dt `Date`. The revision date – when the case was registered at CDC.

current_status `character`. CDC's classification, either `"Laboratory-confirmed case"` or `"Probable Case"`. Map it with `revision_levels` (see above).

sex `character`. `"Female"`, `"Male"`, `"Other"`, `"Unknown"` or `"Missing"`.

n `integer`. Number of cases sharing that combination.

Details

Each row is a unique (onset date, specimen date, CDC report date, status, sex) combination with the number of cases `n`.

The three dates

The source file carries four date columns. `cdc_case_earliest_dt` is derived by CDC as the earliest of the others, and equals `onset_dt` for 99.997% of the rows kept here, so it is dropped as redundant. The three that remain are the only chain that runs forward in time, and they map onto the three roles a `tbl_now()` knows about:

`onset_dt` the **event** – symptoms begin.

`pos_spec_dt` the **report** – the first positive specimen is collected, which is when the surveillance system first sees the case.

`cdc_report_dt` the **revision** – the case is registered at CDC with a status.

`current_status` and `revision_levels`

`current_status` is kept in CDC's own words rather than recoded, because translating it is exactly what `tbl_now(revision_levels = )` is for:

```
revision_levels = c(
  "Laboratory-confirmed case" = "confirmed",
  "Probable Case"             = "pending"
)
```

A *probable* case is one that met the clinical and epidemiological criteria without meeting the laboratory-confirmed definition. Every row here has a positive specimen, so "probable" means the specimen was collected and the case was never laboratory-settled – `"pending"` in this package's vocabulary. Note what is **not** there: CDC does not withdraw cases, so `"retracted"` does not occur in this dataset. It is a two-outcome revision process, and code that needs a retraction has to look elsewhere.

The relationship between the outcome and the revision delay is real rather than fabricated: probable cases are registered a median of 2 days after the specimen, laboratory-confirmed ones 4 days.

What was kept

Cases where all three dates are present, correctly ordered (`onset_dt <= pos_spec_dt <= cdc_report_dt`) and falling entirely within 2020 – a self-consistent "as of the end of 2020" snapshot. Rows out of order are data-entry errors; rows missing a date cannot be placed on the chain at all. See `data-raw/covid_us.R` for the exact duckdb aggregation of the 14 GB source file.

The reporting delay is enormous and heavily right-skewed: cases reached CDC not smoothly but in large backlog dumps – a textbook batch-reporting pattern that `diagnose_batches()` and `transport_discriminant()` recover.

Source

Centers for Disease Control and Prevention (CDC), COVID-19 Response. *COVID-19 Case Surveillance Public Use Data* (version date: June 21, 2024). https://data.cdc.gov/Case-Surveillance/COVID-19-Case-Surveillance-Public-Use-Data/vbim-akqf/about_data. COVID-19 case surveillance data are collected by jurisdictions and reported voluntarily to CDC.

See Also

tbl_now() to declare the date columns; add_revision_date() to attach the third one to an object that has none; revised_cases to count the outcomes; summary() and diagnose() to inspect the result; the package's other datasets – denguedat, mpoxdat, flusight, covid_colombia and hai_bucaramanga.

Examples

```
library(dplyr)

data(covid_us)

# The three dates with CDC's labels translated to this package's vocabulary.
covid_us <- covid_us |>
  filter(onset_dt <= as.Date("2020-02-01"))

tn3 <- tbl_now(
  covid_us,
  event_date       = onset_dt,
  report_date      = pos_spec_dt,
  revision_date    = cdc_report_dt,
  revision_type    = current_status,
  revision_levels = c(
    "Laboratory-confirmed case" = "confirmed",
    "Probable Case"             = "pending"
  ),
  case_count = n,
  strata     = sex,
  data_type  = "count-incidence",
  verbose    = FALSE
)
```

---

denguedat *denguedat: Dengue fever individual-level reporting data from Puerto Rico*

Description

Surveillance data from CDC Division of Vector-Borne Diseases. 1990-2010 case reporting data included.

Usage

```
data(denguedat)
```

Format

A data frame with 52,987 rows (one per case) and 3 variables:

onset_week `Date`. The week symptoms began – the *event* date.

report_week `Date`. The week the case reached the surveillance system – the *report* date. Always on or after `onset_week`.

gender `character`. `"Male"` or `"Female"`. Synthetic; see the note.

Details

Each row represents a case with the columns indicating the following:

- `onset_week`: the week of symptom onset.
- `report_week`: the week of case report.
- `gender`: the gender of the infected individual (randomly assigned with 0.5:0.5 probability of "Male"/"Female").

Note

Data originally from the `NobBS` package. While `onset_week` and `report_week` correspond to actual observed data the `gender` was constructed exclusively for the examples of `NobBS`. It is a synthetic (simulated) variable and does not correspond to any reality.

References

MCGOUGH, Sarah F., et al. Nowcasting by Bayesian Smoothing: A flexible, generalizable model for real-time epidemic tracking. PLoS computational biology, 2020, vol. 16, no 4, p. e1007735.

See Also

`tbl_now()` to declare the date columns; summary() and `diagnose()` to inspect the result; the package's other datasets – denguedat, mpoxdat, flusight, covid_colombia, covid_us and hai_bucaramanga.

Examples

```
data(denguedat)
head(denguedat)

# The two dates every nowcast needs. Weekly data, twenty years of it.
range(denguedat$onset_week)

# Declaring them turns the data frame into a tbl_now.
dengue <- tbl_now(denguedat,
  event_date = onset_week, report_date = report_week,
  strata = gender, verbose = FALSE
)
dengue

# Most cases arrive within a week or two of onset; a few take much longer.
summary(as.numeric(dengue$.delay))
```

diagnose *Diagnose a* tbl_now

Description

[Experimental]

diagnose() is a structural health check. It looks for the things that make a nowcast wrong before any model is fitted – dates out of order, missing values, repeated rows, units that disagree, data after now, event dates too recent to be complete – and returns them as a tibble of findings, sorted worst first.

It is **deterministic and runs no statistical test.** Whether the reporting delay drifts, and whether reports arrive in batches, are questions about a *distribution*, not about the object's structure, so diagnose() leaves them to diagnose_drift() and diagnose_batches() rather than quietly running a test whose method, window and multiplicity correction you did not choose.

Every block is also available on its own – see nowcast_diagnose_components – and diagnose() is exactly the dplyr::bind_rows() of those pieces.

Usage

```
diagnose(x, ...)

## Default S3 method:
diagnose(x, ...)

## S3 method for class 'tbl_now'
diagnose(
  x,
  ...,
  checks = NULL,
  by_strata = NULL,
  strata = NULL,
  warn_non_uniqueness = TRUE
)
```

Arguments

x	A tbl_now object.
...	Unused, for extensibility.
checks	Character vector of checks to run, a subset of c("declarations", "ordering", "missing", "duplicates", "units", "negatives", "now", "truncation", "strata"). Defaults to all of them.
by_strata	Logical. Add one set of rows per stratum, for the checks that are naturally per-stratum (missingness, negative increments, right-truncation, the gap to now). Defaults to TRUE when the object has strata. The checks that are statements about the object as a whole (declarations, units, duplicates, ordering) are always reported once, with stratum = "all".

`strata` Character vector of columns to stratify by. Defaults to `get_strata(x)`.

`warn_non_uniqueness`
Logical. Run the duplicate-row check. Defaults to `TRUE` here, unlike `validate_tbl_now()`, where it defaults to `FALSE` because it runs on every `dplyr` verb.

Value

A tibble with the columns described above, sorted worst first.

The columns

Every function in this family returns the same schema, so results can be stacked with `dplyr::bind_rows()` and filtered with `dplyr::filter()`.

`check` Which block the row belongs to: `"declarations"`, `"ordering"`, `"missing"`, `"duplicates"`, `"units"`, `"negatives"`, `"now"`, `"truncation"` or `"strata"`.

`scope` What the row is about: a column name, a time axis, a pair of axes, or `"all"`.

`stratum` Which subset of the data the row describes: `"all"` for the pooled rows, or the stratum label otherwise.

`status` An **ordered factor**, worst first, so the tibble sorts itself: `error` > `warning` > `note` > `ok` > `skipped`. See the section below.

`n_affected` How many rows (or cases, or dates) the finding is about.

`n_total` How many were considered.

`prop` `n_affected` / `n_total`.

`message` One human sentence, already formatted.

`hint` What to do about it, or `NA`.

`rows` A list-column of offending row indices, so `x[result$rows[[1]], ]` goes straight to the bad rows. Empty when the finding is not about particular rows, or when it was computed on a de-accumulated view whose rows are not the object's own.

What the statuses mean

`error` `validate_tbl_now()` aborts on this. The object is not a usable `tbl_now`.

`warning` `validate_tbl_now()` warns about this.

`note` A `diagnose()`-only observation worth your attention. It is deliberately never promoted to a warning: `validate_tbl_now()` runs on every `dplyr` verb, and a new warning there would turn a quiet construction into a noisy one for data that has always been accepted.

`ok` The check ran and found nothing.

`skipped` Could not be assessed – no revision process, the wrong data type, or an optional package that is not installed.

See Also

nowcast_diagnose_components for the individual blocks; summary() for the descriptive counterpart – what is in the data rather than what is wrong with it; `validate_tbl_now()` for the same findings raised as errors and warnings; `diagnostic_plot()` for the picture version. The *Diagnosing a tbl_now* article goes through the findings one at a time.

Examples

```
data(denguedat)
# The last five years. The full twenty-year series gives the same shape
# of answer, it just takes longer to compute.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
ndata <- tbl_now(recent,
  event_date = "onset_week",
  report_date = "report_week",
  strata = "gender",
  verbose = FALSE
)

# Everything, worst first
diagnose(ndata)

# Only what needs acting on
diagnose(ndata) |> dplyr::filter(status <= "note")

# One block on its own
diagnose(ndata, checks = "units")

## `diagnose()` never stops your pipeline -- it hands back a table for you to
## read. Use validate_tbl_now() when you want a broken object to be an error.
nrow(diagnose(ndata))
```

---

diagnose_batches *Screen the report axis for batched reporting*

Description

[Experimental]

Detects **batches**: report dates at which a stalled reporting system releases a backlog. A batch *moves* reports along the report axis without creating them, so it shows up as a spike preceded by a deficit, while the total over a window spanning both is unchanged. `diagnose_batches()` is completely **model-free** – it needs only a `tbl_now()`, not a fitted model – which makes it the right tool for exploratory data analysis before any nowcasting model is chosen.

Usage

```
diagnose_batches(
  x,
  lookback = 7L,
  baseline_window = NULL,
  period = NULL,
  null_model = c("auto", "poisson", "robust"),
  axis = c("report", "revision"),
  alpha = 0.05,
```

```
  drop_censored = TRUE
)
```

Arguments

x	A tbl_now() object of any data_type.
lookback	Integer k: how many report dates before r the window reaches back. Should comfortably cover the longest plausible stall. Default 7 (a week of daily reporting).
baseline_window	Odd integer width of the smoother used to estimate the baseline (a robust local line, Siegel's repeated median). Must satisfy baseline_window >= 2 * lookback + 3 so that a clean date is never outvoted by a batch episode. Defaults to the smallest admissible odd value (adjusted upward to a multiple of period plus one when period is supplied).
period	Optional integer cycle length of a *scheduled* reporting pattern (e.g. 7 for a weekly cycle on daily data). NULL (default) means the cycle is taken from the object's temporal effects if present (day-of-week -> 7, week-of-year -> 52), and otherwise no calendar correction is applied. A value passed here always wins.
null_model	"auto" (default) picks the null from the data. The exact Poisson/Binomial null assumes Poisson counts *and* a baseline that captures the mean; real surveillance counts are overdispersed, so on non-negative counts auto uses the exact null only when no overdispersion is detected (dispersion at most 1.5) and otherwise falls back to the dispersion-corrected robust normal approximation. Signed (count-cumulative) increments always use the robust null. "poisson" and "robust" force the choice; note that "poisson" is anti-conservative (overflags) on overdispersed counts.
axis	Which time axis to scan for arrivals: "report" (default) or "revision". The question is the same either way – did an unusual number of records land on this date? – so a laboratory clearing its backlog is found exactly as a surveillance system clearing its inbox is. "revision" needs a revision process (see add_revision_date()) and ignores cases that are still "pending", which have no revision date to arrive on.
alpha	Significance level for the Benjamini-Hochberg batch flag. Default 0.05.
drop_censored	Logical. Ignore the rows whose date on axis is flagged censored (is_censored_report, or is_censored_revision on the revision axis). Default TRUE: a censored date is a *bound*, not the date the record arrived, so those rows would pile up on the censoring date and be rediscovered as the very batch the censoring already recorded.

Details

The idea is that a batch **moves** reports along the report axis without creating them. Over a window of report dates that spans both the lull and the release, the *total* is therefore unchanged – every report you would have seen in the window you still see, just on a different day. A genuine surge instead **adds** reports and inflates the window total. So two quantities separate the two cases: the **deficit** (how many reports the days just before the spike were missing) picks up transport, and the window

total (relative to a baseline) picks up creation. A batch has a large deficit but a conserved window total; a surge has an inflated window total but no deficit.

The baseline for each candidate window is refit from report dates lying strictly *outside* that window, using a robust local line (Siegel's repeated median). Smoothing through the episode instead would let the deficit drag the baseline down and the batch would mask itself as a surge.

The `batch` flag is the trustworthy verdict: it compares dates *within* one window (insensitive to the overall level) and is **Benjamini-Hochberg corrected** across every (report date, stratum) pair, so it controls the false discovery rate rather than firing on every point that crosses a raw threshold. A per-point creation ("surge") label is deliberately *not* returned: it only compares the window total against the baseline, so on a steeply curved epidemic curve it fires on ordinary growth. If you need genuine surges, fit a model. For `"count-cumulative"` data the increments are signed and `reported` can be negative (a down-revision).

A reporting system that is always closed at weekends produces every batch symptom, every week, so `diagnose_batches()` needs the length of any scheduled cycle. It reads that from the object's temporal effects when it can: a **day-of-week** effect sets `period = 7`, a **week-of-year** effect `period = 52` (see `add_temporal_effects()`). Pass `period` yourself to override; if the data is daily and carries no temporal effect, the function suggests `period = 7`. With a period set, the baseline is corrected by per-phase medians across cycles, so an irregular batch reads as an excursion relative to the schedule.

Value

A tibble of class `diagnose_batches`, one row per (report date, stratum), with a `print()` method that summarises the flagged dates. Columns:

`report_date` The report (registration) date the row describes.

`stratum` The stratum label, or `"all"` when the data is unstratified.

`reported` Reports recorded on `report_date` (a signed increment for `"count-cumulative"` data, so it can be negative).

`baseline` The robust expected number of reports on `report_date` under "no batch", from the leave-window-out local line.

`deficit` How many reports the `lookback` days *before* `report_date` were missing relative to baseline – the **transport** signal. Large and positive when a stall preceded a spike.

`delta` The window total minus its baseline mean – the **creation** signal. Near zero for a pure batch (mass only moved), large for a surge.

`p_transport` One-sided p-value that the deficit is larger than noise (the raw, per-point transport test).

`p_transport_bh` `p_transport` after a Benjamini-Hochberg correction across all rows; the flag below thresholds this.

`batch` Logical verdict: `TRUE` when `p_transport_bh < alpha` and the window is not still depleted (a hold). This is the column to trust.

See Also

`diagnose_batches2()` for the complementary test on *which* event dates a flagged report date drew from; `transport_discriminant()` for the two coordinates behind the test, without the hypothesis test on top; `simulate_batch()` to plant a known batch and check the screen finds it;

plot_reporting_process() and plot_reporting_triangle() to see it; add_is_censored_report() to record a batch once you believe it. The *Diagnosing a tbl_now* article works through a real example.

Examples

```
data(denguedat)

dengue_tbl <- tbl_now(
  denguedat,
  event_date  = onset_week,
  report_date = report_week,
  data_type   = "linelist",
  verbose     = FALSE
)

# Scan every report date for an unusually large arrival.
screened <- diagnose_batches(dengue_tbl, lookback = 2)
head(screened)

# The dates it flagged, strongest evidence first. Treat these as candidates to
# look into, not as confirmed backlog releases. `reported` against `baseline`
# says how much bigger the arrival was than the surrounding days led you to
# expect.
flagged <- screened[screened$batch, ]
nrow(flagged)
flagged[order(flagged$p_transport_bh), c("report_date", "reported", "baseline")]
```

---

diagnose_batches2 *Test whether one report date drew from unusually old event dates*

Description

[Experimental]

A complement to diagnose_batches(), which sees only report *volumes*. This test asks whether the reports that arrived on a candidate date came from systematically *older* event dates – the signature of a released backlog – by comparing their delays with those of neighbouring report dates. It is **model-free** and, under the conditions below, **exactly distribution-free**.

Usage

```
diagnose_batches2(
  x,
  at,
  neighbours = 3L,
  guard = 1L,
  permute = c("items", "blocks"),
  n_permutations = 999L,
```

```
  axis = c("report", "revision"),
  drop_censored = TRUE,
  seed = NULL
)
```

Arguments

x	A tbl_now() object.
at	The candidate report date (coercible to the class of the report column), typically one flagged by diagnose_batches(). A date on the report grid that carries no rows is reported as zero arrivals, not an error.
neighbours	Number of report dates on each side used as the reference group. Default 3.
guard	Number of report dates immediately either side of at to skip. Default 1. Increase it to at least the longest plausible stall.
permute	"items" (default; exact under log-linear intensity and Poisson counts) or "blocks" (permutes whole report dates; valid under overdispersion).
n_permutations	Number of permutations. Default 999.
axis	Which time axis to scan for arrivals: "report" (default) or "revision". The question is the same either way – did an unusual number of records land on this date? – so a laboratory clearing its backlog is found exactly as a surveillance system clearing its inbox is. "revision" needs a revision process (see add_revision_date()) and ignores cases that are still "pending", which have no revision date to arrive on.
drop_censored	Logical. Ignore the rows whose date on axis is flagged censored (is_censored_report, or is_censored_revision on the revision axis). Default TRUE: a censored date is a *bound*, not the date the record arrived, so those rows would pile up on the censoring date and be rediscovered as the very batch the censoring already recorded.
seed	Optional RNG seed.

Details

The delays of the reports arriving on at are compared with the pooled delays of the reports arriving on nearby dates, using a one-sided rank-sum (Wilcoxon) statistic directed at *longer* delays on at. The p-value comes from a permutation, so no asymptotic approximation is used.

The test is model-free: as long as the epidemic curve is locally smooth, neighbouring report dates share one common delay profile, so their delay labels are exchangeable and the permutation test is (essentially) distribution-free – it needs neither the delay distribution nor the epidemic curve. With Poisson counts permute = "items" is exact; if the counts are overdispersed (neighbouring report dates share event dates, so individual items are not exchangeable) use permute = "blocks", which permutes whole report dates.

at need not be a date that carries rows. A line list cannot represent a zero, so a report date on which nothing arrived has no rows at all; that is the observation "no arrivals", not a missing one, and the test reports zero arrivals for it instead of aborting. Only a date off the object's report grid is an error.

The guard argument omits report dates immediately adjacent to at from the comparison set: if a batch is present, its own deficit dates sit right beside the spike and would contaminate the reference

group. For "count-cumulative" data only positive increments carry a meaningful delay; negative increments (down-revisions) are dropped with a message.

Value

A tibble, one row per stratum, with stratum, n_at, n_reference, mean_delay_at, mean_delay_reference, statistic (standardised rank-sum) and p_value (one-sided: longer delays on at).

See Also

diagnose_batches(), which finds the report dates worth passing to at; simulate_batch() to plant a batch of known shape and check it is recovered; plot_delay_profiles() to see the delay profile this tests.

Examples

```
data(denguedat)

dengue_tbl <- tbl_now(
  denguedat,
  event_date  = onset_week,
  report_date = report_week,
  data_type   = "linelist",
  verbose     = FALSE
)

# Pick a report date to interrogate. A real workflow takes this from
## diagnose_batches(); here we simply name one.
diagnose_batches2(dengue_tbl, at = as.Date("1990-06-25"), n_permutations = 99)

# `n_permutations` sets the resolution of the p-value: 99 keeps the example
## fast, but use the default (999) for anything you intend to report.
```

---

diagnose_changepoint *Detect an abrupt change point in the reporting-delay distribution*

Description

[Stable]

Complements diagnose_drift(). Where that tests for a *gradual* monotonic trend, this tests for a **single abrupt shift** (e.g. a reporting-system change on some date) in the per-period delay summaries, using **Pettitt's** nonparametric change-point test. As with diagnose_drift() it works on both a location statistic (median / mean) and a dispersion statistic (IQR / 10-90 spread), on mature data only, and — being rank-based — it is robust to the skew and serial dependence of a delay series.

Usage

```
diagnose_changepoint(
  x,
  ...,
  stat = c("median", "spread"),
  by_strata = FALSE,
  strata = NULL,
  mature_only = TRUE,
  level = 0.95,
  alpha = 0.05,
  axis = c("report", "revision")
)
```

Arguments

x	A tbl_now object.
...	Passed to the underlying **modifiedmk** function (e.g. nsim for "block-bootstrap").
stat	Which delay summaries to test: any of "median", "mean", "iqr" (q75 - q25) and "spread" (q90 - q10). Defaults to median + spread, i.e. one *location* and one *dispersion* statistic.
by_strata	Logical (default FALSE). When TRUE, the test is run separately per stratum.
strata	Character vector of columns to group on when by_strata = TRUE. NULL (default) uses the object's strata.
mature_only	Logical (default TRUE). Drop event dates after the level incompleteness cutoff before testing.
level	Completeness level for the maturity cutoff (default 0.95).
alpha	Significance level for the drift verdict column (default 0.05).
axis	Which time axis the delay is measured to: "report" (default) or "revision". Report-axis delays are measured from event to report; revision-axis delays are measured from report to revision, the same quantity as .revision_delay. Needs a revision process (see add_revision_date()); cases still "pending" are left out.

Value

A tibble with **one row per requested** stat **per stratum**, and the following columns:

strata character. The stratum the row refers to. When by_strata = FALSE (the default) there is a single stratum labelled "all"; otherwise one level per observed combination of strata.

stat character. Which delay summary was tested — one of "median", "mean", "iqr" or "spread". As in diagnose_drift(), the first two are *location* statistics and the last two *dispersion* statistics.

n integer. **Length of the tested series**: the number of event dates contributing a non-missing value after the mature_only filter — periods, not cases. Series shorter than 8 periods, or with zero variance, are not tested and return NA throughout.

`changepoint` `Date`. The event date of the **last period before the estimated change**; the shift is taken to occur immediately after it. `NA` when the series was too short to test. Note this is reported even when `changepoint_detected` is `FALSE` — Pettitt's test always returns the most extreme candidate split, so this field is only meaningful once the p-value supports it.

`statistic` `numeric`. Pettitt's `K`, the maximum absolute value of the rank statistic `U_t` over all candidate split points. Larger means a cleaner separation between the two sides. It is not standardised, so it grows with `n` and is not comparable across series of different lengths.

`p_value` `numeric`. Two-sided p-value for the null of *no change point*, from the standard approximation $2\exp(-6K^2/(n^3+n^2))$, capped at 1. This approximation is known to be conservative for small `n`.

`before`, `after` `numeric`. The mean of the statistic on each side of `changepoint`, in the object's delay units. These are plain means of the per-period summaries, so they describe the two regimes directly.

`shift` `numeric`. `after` - `before`: the estimated size and direction of the jump, in delay units. Positive means delays got longer after the change point. This is the number to judge operational relevance by.

`changepoint_detected` `logical`. The verdict: `TRUE` when `p_value` < `alpha`. `NA` p-values give `FALSE`.

Interpreting the result

Judge `shift` first and `p_value` second. A statistically detected change point with a `shift` far smaller than the day-to-day noise in the delay series is not worth acting on; a large `shift` is, even at a marginal p-value.

Two structural caveats matter in practice:

- Pettitt's test assumes **exactly one** change point. Given several, it returns the most prominent and silently ignores the rest. If you suspect more, re-run on each side of the first `changepoint` to search recursively.
- A slow monotonic drift will often trip this test too, with the change point landing near the middle of the series. Running `diagnose_drift()` alongside disambiguates: a genuine step shows up here and not necessarily there, while a gradual drift shows up in both.

A confirmed change point usually has an operational explanation — a new laboratory information system, a change in case definition, a reporting mandate, a holiday backlog being cleared. Where it lands is a strong hint about the cause, and about how far back a nowcasting model can safely be fitted: data before the change point comes from a different reporting regime.

Unlike `diagnose_drift()`, this test has no third-party dependency and no meaningful runtime cost, so it is cheap to run routinely.

See Also

`diagnose_drift()` for a gradual trend rather than a single break; `plot_delay_drift()` to see the series and where the break was found; `diagnose_batches()` for a one-day spike rather than a lasting shift.

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
diagnose_changepoint(dengue)
```

`diagnose_drift` *Test whether the reporting-delay distribution drifts over time*

Description

[Stable]

Runs an **autocorrelation-robust monotonic-trend test** on the per-period, count-weighted delay summaries, to answer "do delays drift over time?" in a way that respects the fact that a delay series is correlated with itself.

Usage

```
diagnose_drift(
  x,
  ...,
  stat = c("median", "spread"),
  method = c("hamed-rao", "yue-pilon", "block-bootstrap"),
  by_strata = FALSE,
  strata = NULL,
  mature_only = TRUE,
  level = 0.95,
  alpha = 0.05,
  axis = c("report", "revision")
)
```

Arguments

`x`	A `tbl_now` object.
`...`	Passed to the underlying **modifiedmk** function (e.g. `nsim` for `"block-bootstrap"`).
`stat`	Which delay summaries to test: any of `"median"`, `"mean"`, `"iqr"` (q75 - q25) and `"spread"` (q90 - q10). Defaults to median + spread, i.e. one *location* and one *dispersion* statistic.
`method`	Trend test: `"hamed-rao"` (default; Hamed-Rao variance correction, `modifiedmk::mmkh()`), `"yue-pilon"` (Yue-Pilon, `modifiedmk::mmky()`) or `"block-bootstrap"` (block-bootstrap MK, `modifiedmk::bbsmk()`). See the *Choosing a method* section.
`by_strata`	Logical (default `FALSE`). When `TRUE`, the test is run separately per stratum.
`strata`	Character vector of columns to group on when `by_strata = TRUE`. `NULL` (default) uses the object's `strata`.

mature_only	Logical (default TRUE). Drop event dates after the level incompleteness cutoff before testing.
level	Completeness level for the maturity cutoff (default 0.95).
alpha	Significance level for the drift verdict column (default 0.05).
axis	Which time axis the delay is measured to: "report" (default) or "revision". Report-axis delays are measured from event to report; revision-axis delays are measured from report to revision, the same quantity as .revision_delay. Needs a revision process (see add_revision_date()); cases still "pending" are left out.

Details

For each requested stat (and each stratum) it builds the per-event-date series of that statistic and tests it for a monotonic trend with the **modifiedmk** package, which corrects the Mann-Kendall variance for serial autocorrelation. A plain Mann-Kendall (or an OLS slope) would be anti-conservative here, because positive autocorrelation shrinks the effective sample size.

By default the test uses only **mature** event dates (those on or before the level incompleteness cutoff), because the recent, not-yet-fully-reported dates would otherwise inject a spurious downward trend.

Value

A tibble with **one row per requested** stat **per stratum**, and the following columns:

strata character. The stratum the row refers to. When by_strata = FALSE (the default) there is a single stratum labelled "all"; otherwise one level per observed combination of strata.

stat character. Which delay summary was tested — one of "median", "mean", "iqr" or "spread". "median"/"mean" are *location* statistics (are delays getting longer?); "iqr"/"spread" are *dispersion* statistics (are delays getting more erratic?).

n integer. **Length of the tested series**, i.e. the number of event dates contributing a non-missing value after the mature_only filter. This is a count of *periods*, not a count of cases. Series with n < 10 (or with zero variance) are not tested and return NA for every test column.

tau numeric in [-1, 1]. Kendall's rank correlation between the statistic and time — the *effect size*. Positive means delays are growing, negative means they are shrinking. Roughly, |tau| below 0.1 is a negligible trend even when p_value is small.

sens_slope numeric. Sen's slope: the median pairwise rate of change, expressed **in delay units per period** — so for weekly data with delays measured in weeks, "weeks of delay gained per week elapsed". Multiply by n for the total drift implied across the series. Unlike an OLS slope this is robust to outlying periods.

statistic numeric. The autocorrelation-corrected Mann-Kendall Z score. Under the null it is standard normal, so |Z| > 1.96 corresponds to p_value < 0.05.

p_value numeric. Two-sided p-value for the null hypothesis of *no monotonic trend*, after the serial-correlation correction implied by method. NA when the series was too short or constant.

method character. The method actually used, echoed back so the result is self-documenting when several runs are bound together.

drift logical. The verdict: TRUE when p_value < alpha. NA p-values give FALSE, so a FALSE means "no drift detected" and not necessarily "no drift".

Interpreting the result

Read `tau` and `sens_slope` *before* `p_value`. On long surveillance series a tiny, operationally irrelevant trend will still be highly significant, so `drift = TRUE` on its own is not a reason to act. The question to ask is whether `sens_slope * n` — the total drift implied over the observed window — is large relative to the delays themselves.

The location and dispersion statistics answer different questions and can disagree, which is informative rather than contradictory:

- `median` drifting up, `spread` flat: reporting is uniformly slower.
- `median` flat, `spread` drifting up: the typical case is unaffected but the tail is getting worse — often a subset of reporting sites degrading.
- both drifting up: broad deterioration in reporting timeliness.

A detected drift means a nowcasting model fitted on a **fixed** delay distribution will be biased, because it is averaging over delay regimes that are not exchangeable. Consider a model with a time-varying delay, or fitting only to the recent, homogeneous stretch of data.

Because this is a trend test it will *not* find an abrupt one-off shift; a step change can even cancel out to a non-significant monotonic trend. Pair it with `diagnose_changepoint()`, which is built for exactly that case.

Choosing a method

All three options are Mann-Kendall tests that correct for the serial correlation of a delay series; they differ in what they assume about that correlation, and in cost.

`"hamed-rao"` **(default)** Inflates the Mann-Kendall variance using all *significant* autocorrelation lags of the detrended ranks. It makes no AR(1) assumption, is deterministic, and is effectively instantaneous, so it is the sensible default for a routine diagnostic. Its variance correction is known to be unstable on short series — treat results with `n` below roughly 30 as indicative only.

`"yue-pilon"` Trend-free pre-whitening, which effectively assumes the series is **AR(1)**. That assumption is a poor fit for daily reporting delays, which carry strong day-of-week periodicity, and pre-whitening is known to remove part of the very trend being tested. Offered for comparability with the hydrology literature; rarely the right choice here.

`"block-bootstrap"` Resamples contiguous blocks, so it accommodates arbitrary dependence *within* a block — including weekly periodicity, if the block length covers it. Statistically the most defensible for daily data, and the best cross-check when a `hamed-rao` result is borderline. Two caveats: it is **stochastic**, so call `set.seed()` first for a reproducible p-value, and it is **thousands of times slower** — it scales at roughly the square of the series length, so a multi-year daily series can take many minutes per statistic. Reduce `nsim` (passed through `...`) or restrict to a shorter window before reaching for it.

When a decision matters, run the default first and confirm a borderline result with `method = "block-bootstrap"` on a restricted window.

See Also

diagnose_changepoint() for an abrupt shift rather than a gradual trend; plot_delay_drift() to see the series being tested; censor_reporting_delays_above() once you decide some delays are not to be believed; diagnose() for the structural checks, which this test deliberately sits outside of.

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
diagnose_drift(dengue)
```

---

diagnostic_plot *Diagnostic plots of the reporting process*

Description

[Stable]

Lays out a gallery of complementary views of a tbl_now's reporting process, all aimed at spotting reporting artefacts – especially *batch reporting*. Each view is also available on its own (see **See also**); diagnostic_plot() picks the ones named in panels and combines them with **patchwork**. Selecting a single panel returns it as a plain plot. Every view is facetted by stratum when the tbl_now declares strata.

Usage

```
diagnostic_plot(
  x,
  panels = "all",
  by = c("report", "event"),
  max_delay = NULL,
  ...,
  plotly = FALSE,
  axis = c("report", "revision"),
  size = 1,
  linewidth = 1,
  grid_linewidth = 0.3,
  palette = .tbl_now_palette()
)
```

Arguments

x	A tbl_now() object.
panels	Which panels, "all" (default) or any subset of "reporting", "triangle", "profiles", "delay_drift" and "transport".

by	For the "profiles" panel, one mark per "report" date (default) or per "event" date.
max_delay	Largest delay on the delay-based panels. NULL (default) caps at the delay covering 99% of reported mass.
...	Batch controls (lookback, period, alpha) routed to the "transport" panel.
plotly	If TRUE, return an interactive **plotly** widget (the panels stacked) instead of a static **patchwork**. Default FALSE.
axis	Which time axis the delay is measured on: "report" (default) or "revision". Report-axis delays are measured from event to report; revision-axis delays are measured from report to revision, the same quantity as .revision_delay. Needs a revision process (see add_revision_date()); cases still "pending" are left out.
size	Multiplier on point and label sizes, forwarded to every panel that draws them ("triangle", "transport"). Default 1.
linewidth	Multiplier on data line widths, forwarded to every panel that draws them ("profiles", "delay_drift"). Default 1.
grid_linewidth	Line width of the reference grids the package draws itself – not **ggplot2**'s panel grid. Forwarded to "triangle", "transport" and "delay_drift". Default 0.3.
palette	A named colour palette (see tbl_now_palette()).

Value

A **patchwork** object, or a single plot when one panel is selected (or a **plotly** widget when plotly = TRUE).

See Also

Every panel is also a function of its own: plot_reporting_process() and plot_epidemic_process() (when reports arrived, versus when cases happened), plot_reporting_triangle() (the full event-by-delay grid), plot_delay_profiles() (each date's delay curve), plot_delay_drift() (whether delays are getting longer), plot_transport_discriminant().

Examples

```
data(denguedat)
# The two and a half years around the 1996 and 1997 backlog dumps: enough
# for the transport panel to have something to flag, quick enough to draw.
window <- denguedat[
  denguedat$onset_week >= as.Date("1995-06-01") &
    denguedat$onset_week <= as.Date("1998-01-01"),
]
dn <- tbl_now(window, onset_week, report_week, verbose = FALSE)
diagnostic_plot(dn, panels = c("triangle", "transport"))
```

engine *Specify a nowcasting model and its arguments*

Description

[Stable]

An **engine** is one modelling package plus every argument that package needs. It is what `run_nowcast()` and `nowcast_backtest()` take, and it is the object `nowcast_fit()` and `nowcast_tidy()` dispatch on.

`engine()` is the general constructor and works for any registered method, including one you wrote yourself. The `engine_*()` functions are its package-specific counterparts: each **names** the arguments of the modelling function it drives, so the ones that matter are visible in the signature and at `?engine_nobbs` rather than buried in a `...` that silently swallows a typo. Anything a named argument does not cover still goes through `....`

Usage

```
engine(
  method,
  ...,
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)
```

Arguments

`method`	A single string naming the method, e.g. `"epinowcast"`. Built-in names are matched case-insensitively. See `list_nowcast_methods()`.
`...`	Further arguments for the modelling function, passed through untouched. In `engine()` this is *every* argument; in the `engine_*()` functions it is whatever their named arguments do not already cover.
`min_date`	How much history to fit on. One of • `NULL` (default) – the whole series; • a `Date` – keep event dates on or after it; • a **single number** – keep the last *n* periods before the object's `now`, counted in the object's event units. The number is usually what you want in a `nowcast_backtest()`: `now` moves between fits, and a fixed calendar date would make the fitted window grow as the backtest walks forward, so the last fit would be trained on more data than the first. Trimming is per engine on purpose – `baselinenowcast` and `diseasenowcasting` take a long series in their stride, while `epinowcast` scales with the number of reference dates and is best given a window.

`quantile_levels`
: Numeric vector of probabilities to report the nowcast at. Defaults to `nowcast_quantile_levels()`. It lives on the engine because for some backends it is a **fit-time model argument**, not a way of summarising afterwards. **NobBS** computes exactly the quantiles it is handed in `specs$quantiles` and keeps no draws, so a level it was never asked for cannot be recovered, and **surveillance** reports a fixed set and warns rather than interpolating. The draw-keeping backends – `baselinenowcast`, `diseasenowcasting`, `epinowcast` and `EpiNow2` – answer any level after the fact.

`label`
: Name for this engine in a `nowcast_backtest()` and in the ensemble weights derived from one. Defaults to the method name. Give one when the same package appears twice with different settings, which is the whole reason two `diseasenowcasting` models can be weighted separately.

Value

An object of class `c(method, "nowcast_engine")`.

See Also

`run_nowcast()`, `nowcast_backtest()`, `list_nowcast_methods()`, and the *Adding your own nowcasting model* article for writing a backend of your own.

Examples

```
engine("baselinenowcast", draws = 500)

# Built-in names are matched case-insensitively
engine("nobbs", max_D = 10)

# The same package twice, told apart by `label`
engine("diseasenowcasting", label = "default")
```

`example_engine` *A toy engine for examples*

Description

[Stable]

A deliberately naive nowcasting engine that needs no modelling package. It exists so that the examples in this package can actually run: every real engine depends on **epinowcast**, **NobBS**, **EpiNow2** or another optional package, and an example that cannot run teaches nothing.

Do not nowcast with this. It does not model the reporting delay at all – it reports the counts that have arrived so far and puts a fixed percentage band around them. Because late reports are exactly what it ignores, it under-predicts recent dates by design, which is a useful thing to *see* and a terrible thing to rely on. For real work use one of the engines – engine_baselinenowcast(), engine_epinowcast(), engine_nobbs() and the rest – or write your own.

Usage

```
example_engine(
  ...,
  spread = 0.2,
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)

## S3 method for class 'example'
nowcast_fit(
  engine,
  x,
  ...,
  spread = 0.2,
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## S3 method for class 'example'
nowcast_tidy(engine, fit, x, ..., quantile_levels)
```

Arguments

`...`	Ignored. Present so the engine accepts the same shape of call as the real ones.
`spread`	Non-negative number setting the width of the interval, as a fraction of the point estimate. `0` gives a point mass at the median.
`min_date`	How much history to fit on. One of • `NULL` (default) – the whole series; • a `Date` – keep event dates on or after it; • a **single number** – keep the last *n* periods before the object's `now`, counted in the object's event units. The number is usually what you want in a `nowcast_backtest()`: `now` moves between fits, and a fixed calendar date would make the fitted window grow as the backtest walks forward, so the last fit would be trained on more data than the first. Trimming is per engine on purpose – `baselinenowcast` and `diseasenowcasting` take a long series in their stride, while `epinowcast` scales with the number of reference dates and is best given a window.
`quantile_levels`	Numeric vector of probabilities to report the nowcast at. Defaults to `nowcast_quantile_levels()`. It lives on the engine because for some backends it is a **fit-time model argument**, not a way of summarising afterwards. **NobBS** computes exactly the quantiles it is handed in `specs$quantiles` and keeps no draws, so a level it was never asked for cannot be recovered, and **surveillance** reports a fixed set and warns rather than interpolating. The draw-keeping backends – `baselinenowcast`, `diseasenowcasting`, `epinowcast` and `EpiNow2` – answer any level after the fact.

label	Name for this engine in a nowcast_backtest() and in the ensemble weights derived from one. Defaults to the method name. Give one when the same package appears twice with different settings, which is the whole reason two diseasenowcasting models can be weighted separately.
engine	An engine() object – the modelling package plus its arguments. S3 dispatch is on its class, so a backend for "mypackage" is a function called nowcast_fit.mypackage(). The whole engine arrives, not just its name, so engine$args, engine$label and the rest are available to a backend that wants them.
x	A tbl_now object.
verbose	Logical. Whether the backend (and the converters feeding it) should be chatty.
fit	The object returned by nowcast_fit().

Details

For each event date (and stratum) it takes the cumulative count reported by now, from get_latest_reported_cases(), and reports that as the median. The other quantile levels are that median scaled linearly by spread, so the 2.5% and 97.5% levels sit at roughly 1 -/+ spread times it.

No random numbers are involved, so it gives the same answer every time and does not disturb the RNG stream.

Value

A nowcast_engine object, as engine() returns, that run_nowcast() and nowcast_backtest() accept.

See Also

nowcast_engines for the engines you would actually nowcast with; engine() for the general constructor; nowcast_fit() and nowcast_tidy(), the two methods this implements – read its source for the shortest possible complete backend. The *Adding your own nowcasting model* article walks through writing a real one.

Examples

```
data(denguedat)
recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
dengue <- tbl_now(recent,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

# It is an ordinary engine, so it goes where a real one goes.
example_engine()

nc <- run_nowcast(dengue, example_engine(), verbose = FALSE)
nc

## `spread` controls how wide the (made-up) interval is.
run_nowcast(dengue, example_engine(spread = 0.5), verbose = FALSE)
```

`flusight` *flusight: NHSN Weekly Hospital Respiratory Data from FluSight*

Description

FluSight's weekly hospital admission prediction targets based on the 'total number of new hospital admissions of patients with confirmed influenza captured during the reporting week' reported through CDC's NHSN (the dataset formerly known as HHS-Protect), Weekly Hospital Respiratory Data. Data was downloaded on November 12th 2025.

Usage

```
data(flusight)
```

Format

A data frame with 452,567 rows and 4 variables:

as_of `Date`. The date this row's value was published – the *report* date. The same week appears many times, once per publication.

target_end_date `Date`. The week being reported on – the *event* date.

location_name `character`. US state or territory.

observation `numeric`. Hospital admissions reported for that week as of that publication date.

Details

Data represents how many cases were considered influenza during the week of *target_end_date* given the information known until week *as_of*. Note that *as_of* is always one week ahead of *target_end_date*.

This is count data with 452,567 rows and 4 columns:

- `as_of`: The report date – the date the snapshot was taken, i.e. what was known as of that week.
- `target_end_date`: The event date – the week the admissions occurred.
- `location_name`: State, district or territory (53 levels).
- `observation`: Case counts for those dates. `NA` for 1,152 rows.

Together, `as_of`, `target_end_date` and `location_name` form a unique key.

Duplicate rows removed

The upstream FluSight `time-series.csv` ships exact duplicate rows – 39,139 of them in this snapshot. They were dropped with `dplyr::distinct()` before the dataset was saved (issue #25), taking it from 491,706 to 452,567 rows.

The removal is lossless: every repeated (`as_of`, `target_end_date`, `location_name`) key carried an *identical* `observation`, with no conflicting values anywhere in the file, so no information was discarded and the key became unique. If you download the upstream file yourself you will still see the duplicates and should `distinct()` them before use.

References

Target data from Flusight. Online: https://github.com/cdcepi/FluSight-forecast-hub/blob/main/target-data/time-series.csv

See Also

tbl_now() to declare the date columns; summary() and diagnose() to inspect the result; the package's other datasets – denguedat, mpoxdat, flusight, covid_colombia, covid_us and hai_bucaramanga.

Examples

```
data(flusight)
head(flusight)

## This is count data: one row per (week, publication date, state).
nrow(flusight)
length(unique(flusight$location_name))

# One state is enough to see the reporting process.
texas <- flusight[flusight$location_name == "Texas", ]
flu <- tbl_now(texas,
  event_date = target_end_date, report_date = as_of,
  case_count = observation, verbose = FALSE
)
flu

# `as_of` is not always the same weekday, so some delays are not whole weeks.
## `align_weeks()` fixes that.
mean(flu$.delay != round(flu$.delay))
```

---

get_latest_first *Cases at a chosen point in the reporting process*

Description

[Stable]

The same event date has more than one count, depending on when you look. A week of dengue onsets might show 12 cases the day reporting starts, 40 a week later, and 47 once everything has arrived. These functions let you pick which of those numbers you want.

Usage

```
get_latest_reported_cases(x, type = "total")

get_initial_reported_cases(x, type = "total")

get_nth_reported_cases(x, delay, type = "total")
```

Arguments

x	A tbl_now object.
type	Which cases to count. One of: "total" (default) every case, whatever the outcome. On the revision axis that means every case that has been settled at all. "confirmed", "retracted", "pending" only the cases with that outcome. "pending" is a reporting-axis question only – a pending case has no revision date – and the revision getters refuse it. "unknown" the cases whose revision_type is NA: settled, but the data does not say which way. "net" confirmed **minus** retracted – the running total as a surveillance system publishes it, which can go **down** when a case is withdrawn. This is the quantity a count-cumulative stream actually reports, and the one **diseasenowcasting**'s cumulative signed-change likelihood is built for. "by_type" one row per outcome instead of one number: the outcome column joins the keys, so you get pending, confirmed and retracted side by side. On an object with no revision process anything but "total" warns and pools, because there is no outcome to filter on.
delay	A single non-negative number (or Inf) giving the maximum reporting delay, in event units, to include. Only used by get_nth_reported_cases().

Details

- get_initial_reported_cases() – the count as **first** seen: the earliest report for that event date. This is what a dashboard would have shown you at the time, and it is always an undercount.
- get_latest_reported_cases() – the count as **latest** seen: the most recent report. This is the current best estimate of what really happened, and it is what you score a nowcast against.
- get_nth_reported_cases() – the count accumulated **within a given delay**. **[Stable]** delay = 0 gives the cases reported on the event date itself, delay = 1 adds those reported one period later, and so on. delay = Inf is the same as get_latest_reported_cases().

The gap between the first and the latest count *is* the reporting delay problem that nowcasting exists to solve.

Value

A count-cumulative tbl_now with one row per event date (and stratum, and grouping column), containing:

- the event-date column – when the cases happened. Its numeric version is .event_num.
- the report-date column – the report that was selected for that event date. Its numeric version is .report_num.
- n – the number of cases reported for that event date at the selected point.
- .delay – the delay of the selected report.

- any strata, covariate, censoring indicator and temporal-effect columns the object carried, plus the caller's grouping columns.

The **revision** columns are not carried: the count pools over many revision dates, so the result has no single one and does not pretend to. `type = "by_type"` is the exception – it keeps the outcome column, declared as a covariate, because that is the whole point of the call and an undeclared column is one `to_count()` would pool away. Use get_latest_revised_cases() when you want the third date on the result.

Grouping is respected

Unlike `to_count()`, these functions **keep the caller's grouping** and answer by it: the grouping columns join the event date and the strata as keys, and come back on the result. That is what lets you ask for the latest count by a **covariate** – a column that matters but is not something you nowcast by – which grouping is the only way to express.

They can do this because they *select* a point in the process rather than reshaping the object: one row in is still one case (or one cell) out. `to_count()` cannot, and warns instead.

See Also

get_latest_revised_cases() and friends for the same idea on the revision process; `to_count()` for the underlying data shapes; `score_nowcast()`, which uses the latest counts as truth.

Examples

```
data(denguedat)
# The last five years. The counters work the same on the full twenty-year
# series, they just have more weeks to walk.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
dengue <- tbl_now(recent,
  report_date = "report_week",
  event_date = "onset_week",
  strata = "gender",
  verbose = FALSE
)

# What the surveillance system showed the very first time it reported each
# week -- an undercount, because the late reports had not arrived yet.
first <- get_initial_reported_cases(dengue)
first

# What it shows now, after all the corrections.
latest <- get_latest_reported_cases(dengue)
latest

# The difference between them is what a nowcast tries to predict.
sum(latest$n) - sum(first$n)

# Everything known within two weeks of onset.
get_nth_reported_cases(dengue, delay = 2)
```

```
# A grouping is answered by, not dropped.
dengue |>
  dplyr::group_by(gender) |>
  get_latest_reported_cases() |>
  dplyr::group_vars()
```

`hai_bucaramanga` *Healthcare-Associated Infections – Bucaramanga, Colombia 2020-2023*

Description

A line list of healthcare-associated infections (IAAS, *Infecciones Asociadas a la Atencion en Salud*) notified in the municipality of Bucaramanga, Santander, Colombia, between January 2020 and January 2023 as reported by March 19th 2026. Each row is one notified infection: a specimen taken from a hospitalised patient, the laboratory result, and the isolated microorganism.

Usage

```
hai_bucaramanga
```

Format

A tibble with 1,423 rows and 13 variables:

id `integer`. The source's `Orden` autonumber. Documented as unique but repeated for 100 records (see above), so it is *not* a reliable key.

specimen_date `Date`. The event date – when the sample was taken from the patient. `NA` for 318 records.

received_date `Date`. When the laboratory received the sample. `NA` for 1,194 records and absent after 2019-12-01.

report_date `Date`. The report date – when the laboratory result was issued. `NA` for 583 records.

specimen `factor`, 6 levels. Sample type: `"Whole blood"`, `"Urine"`, `"Secretions"`, `"Other sterile fluids"`, `"Sputum"`, `"Bronchoalveolar lavage"`. `NA` for clinically-confirmed cases.

test `factor`, 7 levels. Laboratory test performed, e.g. `"Blood culture"`, `"Urine culture"`. `NA` for clinically-confirmed cases.

microorganism `character`, 85 distinct values. The isolated organism, in conventional binomial form, e.g. `"Klebsiella pneumoniae"`. `NA` for clinically-confirmed cases. Where the source recorded only a genus the value is `"<Genus> spp."`.

sex `factor`. `"Female"` or `"Male"`.

age_group `ordered factor`, 11 levels from `"<1"` to `"70+"`.

case_type `factor`. `"Laboratory-confirmed"` (1,090) or `"Clinically-confirmed"` (333).

final_condition `factor`. Patient status on discharge: `"Alive"` or `"Dead"`. `NA` for 8 records.

icu_type `factor`. Intensive-care unit where the case occurred: `"Adult"`, `"Paediatric"` or `"Neonatal"`.

institution `integer`, 1-10. Anonymised code of the reporting health institution.

Details

In the nowcasting context the **event date** is specimen_date (when the sample was taken from the patient) and the **report date** is report_date (when the laboratory result was issued). The delay between the two is the specimen-to-result turnaround that a nowcasting model would estimate and correct for. A third date, received_date, records when the laboratory received the sample and splits the delay into a transport and a processing leg – but see the warnings below before relying on it.

The column names and all categorical values have been translated from the original Spanish. The dataset has been trimmed to the columns relevant to a delay or nowcasting analysis; see data-raw/hai_bucaramanga.R in the package sources for the full translation tables and the list of dropped columns.

Data quality – read this first

This is a real, unpolished open-data extract, and it is included partly *because* it is messy: it is a realistic exercise for the delay diagnostics in this package (diagnose_drift(), diagnose_changepoint(), plot_delay_profiles()). Nothing below has been silently repaired.

Missing dates The source uses 1900-01-01 as an undocumented missing-date sentinel. It has been converted to NA here – left in, it produces delays of about -45,000 days. After conversion specimen_date is present for 1,105 records (77.7%), report_date for 840 (59.0%) and received_date for only 229 (16.1%).

received_date **is largely unusable** Beyond being 84% missing, it stops entirely at 2019-12-01, so the three-date chain exists only for the first third of the study period. Prefer the specimen_date -> report_date pair.

Negative delays 88 records (10.7% of the otherwise-valid pairs) have a report_date *before* their specimen_date, by up to 331 days. These are kept exactly as recorded. Filter them out before fitting anything.

Exact duplicates 100 records are byte-identical duplicates of another record, sharing the id that the source documents as a unique autonumber. All 1,423 rows are shipped for fidelity; use dplyr::distinct() to reduce to the 1,323 unique records.

Clinically-confirmed cases For the 333 cases confirmed on clinical grounds rather than by laboratory, the source wrote the literal string "CONFIRMADO POR CLINICA" into the muestra, nombre prueba and microorganismo fields. Those are NA here; the information is preserved in case_type.

Sparsity Only 738 records (51.9%) support a non-negative specimen_date -> report_date delay, spread over 488 distinct event dates – roughly 1.5 cases per event date. That is thin for fitting a nowcasting model, though ample for delay diagnostics. Aggregating to weeks or months is usually necessary.

The delay distribution is strongly bimodal: the median is 3 days but the 90th percentile is 92 days, which makes this a useful test case for delay diagnostics that assume a unimodal delay.

Source

Secretaria de Salud y Ambiente de Bucaramanga, via the Colombian national open-data portal https://www.datos.gov.co/Salud-y-Protecci-n-Social/48-Infecciones-asociadas-a-la-atenci-n-en-salud-w4zx-wbff/about_data. Retrieved 2026-08-17. Column names and categorical values translated from Spanish; see data-raw/hai_bucaramanga.R for the mapping.

See Also

`tbl_now()` to declare the date columns, and add_revision_date() for the third one this dataset has; `diagnose()` and `diagnose_drift()`, which this dataset is deliberately messy enough to exercise; the package's other datasets – denguedat, mpoxdat, flusight, covid_colombia and covid_us.

Examples

```
data(hai_bucaramanga)

# The source ships 100 exact duplicates and a number of unusable rows.
# Reduce to unique records with a valid, non-negative reporting delay.
iaas_clean <- dplyr::distinct(hai_bucaramanga) |>
  dplyr::filter(
    !is.na(specimen_date), !is.na(report_date),
    report_date >= specimen_date
  )
nrow(iaas_clean)

# Roughly 1.5 cases per event date, so aggregate to weeks before building a
# tbl_now for anything model-shaped.
iaas_now <- tbl_now(
  iaas_clean,
  event_date  = "specimen_date",
  report_date = "report_date",
  verbose     = FALSE
)
iaas_now

# The delay is strongly bimodal: a 3-day median with a long secondary mode.
quantile(iaas_now$.delay, c(0.5, 0.75, 0.9, 0.99), na.rm = TRUE)
```

---

is_nowcast_engine *Is this an engine?*

Description

[Stable]

Tests whether an object is a nowcasting engine – the specification built by `engine()` that says which modelling package to use and how to configure it. A bare package name is *not* an engine, which is what this is usually used to check.

Usage

```
is_nowcast_engine(x)
```

Arguments

x	An object.

Value

A single TRUE or FALSE.

See Also

engine() to build one; nowcast_engines for the engines that ship with the package; run_nowcast(), which takes one.

Examples

```
## Built by engine(): yes.
is_nowcast_engine(engine("baselinenowcast"))

# The name of a package on its own: no.
is_nowcast_engine("baselinenowcast")
```

---

is_tbl_nowcast *Is this object a* tbl_nowcast*?*

Description

[Stable]

Tests whether an object is a fitted nowcast – the thing run_nowcast() returns – rather than the data a nowcast is fitted to (for which see is_tbl_now()).

Usage

```
is_tbl_nowcast(x)
```

Arguments

x	An object.

Value

A single TRUE when x is a tbl_nowcast, FALSE otherwise.

See Also

tbl_nowcast for the class itself; run_nowcast() which produces one; is_tbl_now() for the input side.

Examples

```
# A number is not a nowcast.
is_tbl_nowcast(1)

## The object run_nowcast() returns is.
predictions <- data.frame(
  onset_week = as.Date("2020-01-05"),
  .quantile_level = c(0.5, 0.9), .value = c(10, 14)
)
nc <- tbl_nowcast(
  predictions = predictions, method = "toy", event_date = "onset_week"
)
is_tbl_nowcast(nc)
```

---

is_weekday *Is a date a weekday or a weekend?*

Description

[Stable]

Reporting almost always slows down at the weekend, which is one of the strongest and most predictable patterns in surveillance data. This tells you which days are which, and lets you say what "weekend" means – it is Friday and Saturday in much of the Middle East, and Sunday alone in some countries.

Usage

```
is_weekday(date, weekend_days = c("Sat", "Sun"))
```

Arguments

`date`	A Date (or POSIXt) object. May be a vector.
`weekend_days`	A character or numeric vector defining which days count as the weekend. Defaults to Saturday and Sunday. • Character: day names or abbreviations, case-insensitive – `c("Mon", "Tuesday", "wed", ...)`. English names always work, whatever the session's locale is, and so do the names of the current locale, with or without their accents – under `LC_TIME = "es_ES"` both `c("Sat", "Sun")` and `c("sáb", "dom")` mean the weekend. • Numeric: integers 1-7 in `lubridate::wday()` numbering with `week_start = 1`, so **1 = Monday** and 7 = Sunday.

Value

A logical vector, `TRUE` where the date is a weekday and `FALSE` where it falls on the weekend.

See Also

`temporal_effects()` and `add_temporal_effects()`, which use this to build the day-of-week and weekend terms a model can fit; plot_day_of_week_effects() to see the effect in the data; `align_weeks()`, whose `align_on_day` uses this same ISO numbering.

Examples

```
is_weekday(as.Date("2020-04-22")) # TRUE (Wed)
is_weekday(as.Date("2020-04-19")) # FALSE (Sun)

## Middle East weekend (Fri - Sat)
is_weekday(as.Date("2020-04-17"), weekend_days = c("Fri", "Sat"))

# Weekend only on Friday
is_weekday(as.Date("2020-04-17"), weekend_days = "Friday")
is_weekday(as.Date("2020-04-18"), weekend_days = "Friday")

## Weekend on Sun - Mon (numeric: 7 = Sun, 1 = Mon)
is_weekday(as.Date("2020-04-20"), weekend_days = c(7, 1))

## Day names of the session's own locale are understood too
locale_weekend <- format(as.Date(c("2020-04-18", "2020-04-19")), "%a")
is_weekday(as.Date("2020-04-18"), weekend_days = locale_weekend)
```

`list_nowcast_methods` *List the available nowcasting methods*

Description

[Stable]

Scans the S3 methods registered for `nowcast_fit()` in every loaded namespace, so any backend you (or another package) defined shows up here as soon as it is loaded.

`"example"` in the list is `example_engine()`, the toy used to keep this package's examples runnable. It is not a nowcasting method – ignore it when choosing one.

Usage

```
list_nowcast_methods(installed_only = TRUE)
```

Arguments

`installed_only`	Logical. When `TRUE` (default) only the methods whose backing package is installed are returned. Set to `FALSE` to see every registered method.

Value

A character vector of method names, suitable for passing to `engine()`.

See Also

`engine()` to turn one of these names into a configured model; nowcast_engines for the engines that ship with the package; `nowcast_fit()` and `nowcast_tidy()`, the two functions a new method must define. The *Adding your own nowcasting model* article walks through writing one.

Examples

```
## What can this installation actually fit right now? ("example" is the toy
# engine, not a method you would nowcast with.)
list_nowcast_methods()

# Including methods whose modelling package is not installed
list_nowcast_methods(installed_only = FALSE)
```

`mpoxdat` *mpoxdat: Mpox reporting data from the 2022 New York City outbreak*

Description

Surveillance line list data provided by the New York City (NYC) Health Department at https://github.com/nychealth/mpox_no to accompany a nowcasting performance evaluation (doi: 10.2196/56495). Patients with a confirmed or probable mpox diagnosis or illness onset from July 8 through September 30, 2022 were included. The original dataset was aggregated and pre-processed as described in the note below.

Usage

```
data(mpoxdat)
```

Format

A data frame with 1,417 rows and 4 variables:

dx_date `Date`. Specimen collection date of the first positive result – the *event* date.

dx_report_date `Date`. When the Health Department received that result – the *report* date.

race `character`. Synthetic; see the note.

n `integer`. Number of cases with that combination.

Details

This is **count** data: each row holds the number of cases sharing a diagnosis date, a report date and a race. The columns are as follows:

- `dx_date`: is the specimen collection date of the first positive mpox laboratory result,
- `dx_report_date`: is the date the report of first positive mpox laboratory result was received by the NYC Health Department,
- `n`: the case count of individuals within those dates.

- race: the race corresponding to those cases. Race was randomly assigned with probabilities "Non-Hispanic White" = 0.309, "Hispanic" = 0.283, "Black" = 0.202, "Asian" = 0.156, and "Other" = 0.05 which follow what has been reported for the US Census.

Note

While dx_date, dx_report_date and n correspond to actual observed data the race was constructed exclusively for the examples of this package. It is a synthetic (simulated) variable and does not correspond to any reality.

References

ROHRER, Rebecca, et al. Nowcasting to Monitor Real-Time Mpox Trends During the 2022 Outbreak in New York City: Evaluation Using Reportable Disease Data Stratified by Race or Ethnicity. Online Journal of Public Health Informatics, 2025, vol. 17, no 1, p. e56495.

See Also

tbl_now() to declare the date columns; summary() and diagnose() to inspect the result; the package's other datasets – denguedat, mpoxdat, flusight, covid_colombia, covid_us and hai_bucaramanga.

Examples

```
data(mpoxdat)
head(mpoxdat)

# Count data, and daily rather than weekly -- unlike denguedat.
mpox <- tbl_now(mpoxdat,
  event_date = dx_date, report_date = dx_report_date,
  case_count = n, strata = race, verbose = FALSE
)
mpox

# A short, sharp outbreak: about three months of data.
range(mpoxdat$dx_date)
sum(mpoxdat$n)
```

---

nowcast_backtest *Refit several methods at past* now *dates and score them*

Description

[Stable]

Walks back through time: for every date in now_dates, the tbl_now is truncated to the reports that were available then, each method is refitted on that snapshot, and the resulting nowcast is scored against the resolved truth defined by truth_axis and truth_type (reported totals by default). This is what turns a set of models into ensemble weights (see nowcast_weights() and nowcast_ensemble()).

Be aware that this refits every model once per date: with Bayesian backends and a long now_dates it is genuinely expensive.

Usage

```
nowcast_backtest(
  x,
  ...,
  now_dates = NULL,
  horizon = 4,
  n_dates = 4L,
  seed = NULL,
  keep_draws = FALSE,
  on_error = c("warn", "abort"),
  verbose = TRUE,
  truth_axis = c("report", "revision"),
  truth_type = "total"
)
```

Arguments

x	A `tbl_now` object holding the *full* data (the later reports or revisions are what the retrospective nowcasts are scored against). Covariates on each retrospective snapshot should mean values available as of that snapshot's `now`; do not attach future realized covariate values unless they are an explicit forecast input for the engine.
...	The `engine()` objects to backtest, one per model. Each carries its own arguments, so there is no keyed side-table of per-method options to get wrong. Give an engine a `label` (or name the argument) when the same package appears twice: `engine_diseasenowcasting(label = "ar1", model = ...)` and a plain `engine_diseasenowcasting()` are backtested separately, so `nowcast_weights()` can learn a weight for each – matching how `nowcast_ensemble()` takes a named list of members. An engine with no label is labelled by its method.
now_dates	Vector of retrospective nowcast origins. Defaults to the `n_dates` most recent report-axis dates that are at least `horizon` units before the object's `now`; these dates are used as as-of origins, not as a filter on target event dates.
horizon	Number of time units of hindsight required when `now_dates` is chosen automatically. Default `4`.
n_dates	Number of automatic retrospective origins. Default `4`. Ignored when `now_dates` is supplied explicitly.
seed	Optional integer. When given, the RNG is seeded **immediately before each fit**, from `seed` and the label and date that fit is for. One `set.seed()` before the whole backtest is not enough: it only pins anything if every method consumes the same random numbers in the same order, so dropping a method, or refitting one date, silently moves every other fit. Seeding per (label, date) makes a fit depend only on which fit it is.
keep_draws	Logical. Whether to retain every posterior draw from every successful fit. Default `FALSE`, because this can make a backtest much larger. Set it to `TRUE` when the backtest should be passed directly to `scoringutils::as_forecast_sample()`. Engines that return only quantiles still cannot be converted to samples.

on_error	Either "warn" (default) to skip a model/date that fails with a warning, or "abort" to stop.
verbose	Logical. Whether to report progress.
truth_axis	Which process defines the observed counts. "report" (default) scores counts eventually reported. "revision" scores counts eventually resolved on the revision axis and requires a revision-aware truth.
truth_type	Which case type to score. Defaults to "total". Revision types such as "confirmed", "retracted", "pending", "unknown" and "net" follow the same meanings as get_latest_reported_cases() and get_latest_revised_cases(). "by_type" is refused because scoring needs one observed value per event-date/stratum target.

Value

An object of class nowcast_backtest: a list with

scores A tibble of per-date scores with an extra .now column.

predictions A tibble of every retrospective quantile prediction.

draws When keep_draws = TRUE, a tibble of the retained draws; otherwise NULL.

timings A tibble with one row per attempted engine/date fit, its elapsed time in seconds, whether it succeeded, and any error text.

truth The observed counts used for scoring.

methods The labels that produced at least one nowcast.

now_dates The dates that were nowcast.

Use the result directly with scoringutils

A nowcast_backtest has methods for scoringutils::as_forecast_quantile(), scoringutils::as_forecast_point and scoringutils::as_forecast_sample(), so no manual reshaping is needed:

```
quantile_forecast <- scoringutils::as_forecast_quantile(bt)
point_forecast <- scoringutils::as_forecast_point(bt)
```

For sample forecasts, create the backtest with keep_draws = TRUE and use scoringutils::as_forecast_sample(bt). The returned forecast objects can be passed to any compatible scoringutils workflow. For example, relative WIS is obtained with:

```
relative_scores <- quantile_forecast |>
  scoringutils::score() |>
  scoringutils::add_relative_skill(metric = "wis")
```

model, now, the event-date column, and declared strata are retained as forecast units, allowing scores to be extended, regrouped, or summarised without returning to the internal tbl.now representation.

Every engine must report the same quantile levels

A backtest exists to compare models, and two models summarised at different levels are not comparable: the weighted interval score is an average over the levels reported, so a model asked for three of them and one asked for nine are scoring different quantities. Mismatched engines are therefore an **error** rather than a warning.

This matters most for the engines where the levels are a *fit-time* argument. **NobBS** computes exactly the quantiles it is handed and keeps no draws, so a level it was never asked for cannot be recovered afterwards – and an ensemble weighted from such a backtest would silently fall back to whatever levels its members happened to share.

See Also

`engine()` to specify each model being compared, and its `min_date` argument, which matters here because `now` moves between fits; `score_nowcast()` for the scores computed at each `now`; `nowcast_weights()` to turn the result into ensemble weights, and `nowcast_ensemble()` to use them; `scoringutils::score()` and `scoringutils::add_relative_skill()` for an extensible scoring workflow. The *One call, many models* article compares several packages this way.

Examples

```
data(denguedat)

# A short recent window keeps the example quick.
recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
dengue <- tbl_now(recent,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

## `example_engine()` is a toy that ignores the reporting delay entirely; it
# is used here only so the example runs without a modelling package.
## Swap in a real one -- `engine_baselinenowcast()`, `engine_epinowcast()`,
## `engine_nobbs()` -- for anything you intend to act on.

# Refit at two past `now` dates and score each against what is known now.
bt <- nowcast_backtest(dengue,
  example_engine(label = "carry forward"),
  now_dates = as.Date(c("2010-10-04", "2010-11-15")),
  verbose = FALSE
)
head(bt$scores)

# Naming several engines compares them on identical data and dates.
bt$methods

# With a real model the call is the same, with a real engine.
if (requireNamespace("baselinenowcast", quietly = TRUE)) {
  nowcast_backtest(dengue,
    engine_baselinenowcast(draws = 100),
    now_dates = as.Date("2010-11-15"), verbose = FALSE
  )$scores
}
```

nowcast_data_getters *Read what a* tbl_now *was told about itself*

Description

[Stable]

When you build a tbl_now() you tell it which column is the event date, which is the report date, which are strata, and so on. These functions read that information back.

They are how the rest of the package – and any modelling code you write yourself – finds the right columns without hard-coding names. Rather than assuming a column is called onset_week, write x[[get_event_date(x)]] and your code works on any tbl_now.

Usage

```
get_event_date(x)

get_report_date(x)

get_strata(x)

get_num_strata(x)

get_covariates(x)

get_num_covariates(x)

get_now(x)

get_report_units(x)

get_event_units(x)

get_data_type(x)

get_temporal_effects(x)

get_temporal_effect_cols(x)

get_is_censored_report(x)

get_case_count(x)

get_revision_date(x)
```

```
get_revision_type(x)

get_revision_units(x)

get_is_censored_revision(x)

get_revision_levels(x)

has_revision(x)
```

Arguments

x	A tbl_now object.

Details

Most of these return a **column name**, not the column itself. To get the values, index with the name: `x[[get_event_date(x)]]`.

A getter returns `NULL` when the object was never told about that attribute, so `is.null(get_strata(x))` is the test for "unstratified". The two counting helpers, `get_num_strata()` and `get_num_covariates()`, return `0` instead, which is usually easier to work with.

Value

A column name, a count, or a metadata value, depending on the function:

`get_event_date()`, `get_report_date()` Character. The name of the column holding the date the event happened / was reported.

`get_case_count()` Character, or `NULL` for linelist data. The name of the column holding the number of cases.

`get_strata()`, `get_covariates()` Character vector of column names, or `NULL` when there are none.

`get_num_strata()`, `get_num_covariates()` Integer count, `0` when there are none.

`get_is_censored_report()` Character, or `NULL`. The name of the column flagging reports whose date is only an upper bound.

`get_is_censored_revision()` Character, or `NULL`. The same on the revision axis: the column flagging rows whose *revision* delay is a bound rather than a measurement.

`get_now()` The `Date` (or number) the nowcast is anchored on.

`get_event_units()`, `get_report_units()` One of `"days"`, `"weeks"`, `"months"`, `"years"` or `"numeric"` – the grid each date lives on.

`get_data_type()` One of `"linelist"`, `"count-incidence"` or `"count-cumulative"`. See `to_count()`.

`get_temporal_effects()` The `temporal_effects()` specification the object carries, or `NULL`. This is the *request*, not the data.

`get_temporal_effect_cols()` Character vector of the temporal-effect columns actually materialised in the data by `compute_temporal_effects()`; `character(0)` when none have been.

`get_revision_date()`, `get_revision_type()` Character, or `NULL`. The name of the column holding the date a case was resolved, and of the column holding how it resolved.

`get_revision_units()` The grid the revision date lives on, or `NULL` when the object carries no revision process.

`get_revision_levels()` The named dictionary translating the labels in the data into the canonical outcomes, or `NULL` when the column was already canonical.

`has_revision()` `TRUE` when the object carries a revision date. Every code path must work when it is `FALSE`, because most objects have no third date.

The revision process, the optional third date

A `tbl_now` may carry a **third** date beyond the event and the report: the date a case was resolved, either confirmed or retracted. Think of influenza: symptom onset is the event, the medical visit is the report, and the laboratory result is the revision – which can come back negative, in which case the case is *retracted* rather than confirmed.

It is optional and most objects do not have one, so `has_revision()` gates the four getters below it: they all return `NULL` on an object that was never given a third date.

See Also

`tbl_now_attributes()` to get all of them at once; `add()`, change() and remove() to set them, including add_revision_date(); get_latest_revised_cases() and get_latest_revised_cases(type = "net") to count the outcomes; revision_delay for how long resolution takes; get_latest_reported_cases() and friends for reading the counts rather than the metadata.

Examples

```
data(denguedat)
ndata <- denguedat |>
  tbl_now(
    event_date = onset_week,
    report_date = report_week,
    strata = gender,
    t_effects = temporal_effects(month_of_year = TRUE),
    verbose = FALSE
  ) |>
  compute_temporal_effects()

# The two dates every nowcast needs.
get_event_date(ndata)
get_report_date(ndata)

# Use the name to reach the column, so the code does not depend on it.
head(ndata[[get_event_date(ndata)]])

# Strata are groups you want separate nowcasts for; covariates are not.
get_strata(ndata)
get_num_strata(ndata)

## Nothing was declared a covariate, so this is NULL (and the count is 0).
```

```
get_covariates(ndata)
get_num_covariates(ndata)

# Likewise for a censoring indicator that was never supplied.
get_is_censored_report(ndata)
get_is_censored_revision(ndata)

# The as-of moment, and the calendar grid the dates live on.
get_now(ndata)
get_event_units(ndata)
get_report_units(ndata)

# Linelist means one row per case; there is no count column yet.
get_data_type(ndata)
get_case_count(ndata)

## After to_count() there is one.
counts <- to_count(ndata, to = "count-incidence")
get_data_type(counts)
get_case_count(counts)

# The temporal-effects request, versus the columns it actually produced.
get_temporal_effects(ndata)
get_temporal_effect_cols(ndata)

# The third date is optional, so ask before you read it.
has_revision(ndata)
get_revision_date(ndata)

## Once one is attached, the same name-then-index pattern applies.
data(hai_bucaramanga)
hai <- hai_bucaramanga |>
  dplyr::filter(!is.na(specimen_date), !is.na(report_date)) |>
  tbl_now(
    event_date = specimen_date, report_date = report_date,
    data_type = "linelist", verbose = FALSE
  ) |>
  add_revision_date(received_date) |>
  suppressWarnings()

has_revision(hai)
get_revision_date(hai)
get_revision_units(hai)
head(hai[[get_revision_date(hai)]])
```

---

nowcast_diagnose_components

Individual blocks of a tbl_now *diagnosis*

Description

[Stable]

Each function returns one block of `diagnose()`, in the same schema, so they can be stacked with `dplyr::bind_rows()` or used on their own.

- `diagnose_declarations()` – the attributes and the columns they name: types, existence, collisions, columns the object was never told about, and temporal effects that were added but never materialised.
- `diagnose_ordering()` – the `event <= report <= revision` timeline.
- `diagnose_missing()` – `NA` values, per column and per stratum. An `NA` *count* is reported neutrally: in a reporting triangle it means *not yet observed*, which is correct data rather than a defect.
- `diagnose_duplicates()` – rows that repeat on the full key: the declared dates, the revision type, the strata, the covariates and the censoring flags. A row with an `NA` in that key is not compared, because *unobserved* cannot be shown to equal *unobserved*.
- `diagnose_units()` – the declared units against each other, against the calendar the dates actually land on, and against the delay they produce.
- `diagnose_negatives()` – negative counts, and the negative increments a downward revision leaves behind when cumulative data is de-accumulated.
- `diagnose_now()` – anything dated after `now`, and how stale the object is.
- `diagnose_truncation()` – how many recent event dates are still immature, and how much of their eventual total is probably still missing.
- `diagnose_strata()` – the smallest and the sparsest stratum, and the revisions still pending.

Usage

```
diagnose_declarations(x, by_strata = NULL, strata = NULL)

diagnose_ordering(x, by_strata = NULL, strata = NULL)

diagnose_missing(x, by_strata = NULL, strata = NULL)

diagnose_duplicates(
  x,
  by_strata = NULL,
  strata = NULL,
  warn_non_uniqueness = TRUE
)

diagnose_units(x, by_strata = NULL, strata = NULL)

diagnose_negatives(x, by_strata = NULL, strata = NULL)

diagnose_now(x, by_strata = NULL, strata = NULL)

diagnose_truncation(x, by_strata = NULL, strata = NULL)
```

```
diagnose_strata(x, by_strata = NULL, strata = NULL)
```

Arguments

x	A tbl_now object.
by_strata	Logical. Add one set of rows per stratum, for the checks that are naturally per-stratum (missingness, negative increments, right-truncation, the gap to now). Defaults to TRUE when the object has strata. The checks that are statements about the object as a whole (declarations, units, duplicates, ordering) are always reported once, with stratum = "all".
strata	Character vector of columns to stratify by. Defaults to get_strata(x).
warn_non_uniqueness	Logical. Run the duplicate-row check. Defaults to TRUE here, unlike validate_tbl_now(), where it defaults to FALSE because it runs on every dplyr verb.

Value

A tibble in the schema documented in diagnose().

See Also

diagnose(), which stacks all of these and sorts them worst-first; validate_tbl_now() for the same findings raised as errors and warnings; nowcast_summary_components for what *is* in the data rather than what is wrong with it; diagnose_drift(), diagnose_changepoint() and diagnose_batches() for the statistical tests diagnose() deliberately does not run. The *Diagnosing a tbl_now* article explains how to read each finding.

Examples

```
data(denguedat)
# The last five years. The full twenty-year series gives the same shape
# of answer, it just takes longer to compute.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
ndata <- tbl_now(recent,
  event_date = "onset_week",
  report_date = "report_week",
  strata = "gender",
  verbose = FALSE
)

# Is the object described correctly, and do the dates make sense?
diagnose_declarations(ndata)
diagnose_ordering(ndata)
diagnose_units(ndata)
diagnose_now(ndata)

# Is anything missing, repeated, negative, or cut off at the recent edge?
diagnose_missing(ndata)
diagnose_duplicates(ndata)
```

```
diagnose_negatives(ndata)
diagnose_truncation(ndata)

# Are the strata usable?
diagnose_strata(ndata)

## Each returns the same schema, so they stack the way diagnose() stacks them.
dplyr::bind_rows(
  diagnose_units(ndata),
  diagnose_now(ndata)
)
```

`nowcast_engines` *Engines for the built-in nowcasting packages*

Description

[Stable]

One constructor per supported modelling package. Each is `engine()` with the arguments of that package's own entry point spelled out, so the ones that matter are visible in the signature and a typo is an error rather than a silently ignored extra.

The *One dataset, many nowcasts* article documents each package's own API; this page is about driving it through `run_nowcast()`.

Usage

```
engine_diseasenowcasting(
  ...,
  model = NULL,
  type = NULL,
  n_draws = NULL,
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)

engine_baselinenowcast(
  ...,
  draws = 1000,
  delays_unit = NULL,
  max_delay = NULL,
  strata_sharing = "none",
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)
```

```
engine_epinowcast(
  ...,
  preprocess_args = list(),
  expectation = NULL,
  reference = NULL,
  report = NULL,
  fit = NULL,
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)

engine_nobbs(
  ...,
  max_D = NULL,
  moving_window = NULL,
  specs = NULL,
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)

engine_surveillance(
  ...,
  D = NULL,
  when = NULL,
  fit_method = NULL,
  control = NULL,
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)

engine_epinow2(
  ...,
  generation_time = NULL,
  delays = NULL,
  truncation = NULL,
  rt = NULL,
  obs = NULL,
  stan = NULL,
  convert_args = list(),
  min_date = NULL,
  quantile_levels = nowcast_quantile_levels(),
  label = NULL
)
```

Arguments

`...` Further arguments for the modelling function, passed through untouched. In `engine()` this is *every* argument; in the `engine_*()` functions it is whatever their named arguments do not already cover.

`model`, `type`, `n_draws`
(`engine_diseasenowcasting()`) Arguments of `diseasenowcasting::nowcast()`. `model` is where the epidemic and revision processes are chosen, e.g. `diseasenowcasting::model(epide = diseasenowcasting::ar1_epidemic())`. On `count-cumulative` data, diseasenowcasting selects its cumulative model automatically unless you supply one with `diseasenowcasting::model(cumulative = diseasenowcasting::cumulative_process()`

`min_date` How much history to fit on. One of

- `NULL` (default) – the whole series;
- a `Date` – keep event dates on or after it;
- a **single number** – keep the last *n* periods before the object's `now`, counted in the object's event units.

The number is usually what you want in a `nowcast_backtest()`: `now` moves between fits, and a fixed calendar date would make the fitted window grow as the backtest walks forward, so the last fit would be trained on more data than the first. Trimming is per engine on purpose – `baselinenowcast` and `diseasenowcasting` take a long series in their stride, while `epinowcast` scales with the number of reference dates and is best given a window.

`quantile_levels`
Numeric vector of probabilities to report the nowcast at. Defaults to `nowcast_quantile_levels()`.

It lives on the engine because for some backends it is a **fit-time model argument**, not a way of summarising afterwards. **NobBS** computes exactly the quantiles it is handed in `specs$quantiles` and keeps no draws, so a level it was never asked for cannot be recovered, and **surveillance** reports a fixed set and warns rather than interpolating. The draw-keeping backends – `baselinenowcast`, `diseasenowcasting`, `epinowcast` and `EpiNow2` – answer any level after the fact.

`label` Name for this engine in a `nowcast_backtest()` and in the ensemble weights derived from one. Defaults to the method name. Give one when the same package appears twice with different settings, which is the whole reason two `diseasenowcasting` models can be weighted separately.

`draws`, `delays_unit`, `max_delay`
(`engine_baselinenowcast()`) Number of nowcast samples, the unit of the reporting triangle's delay axis (inferred from the object's units when `NULL`), and how many delay periods to keep – `max_delay = 10` keeps delays 0-9, as in `tbl_now_to_baselinenowcast()`. The last one is not only about speed: `baselinenowcast` needs more reference dates than delay columns, so a **snapshot ("as of") series** – which re-reports every past period in every snapshot, and therefore has a delay axis as long as the series itself – cannot be fitted at all until the axis is capped. The error says which number to use.

`strata_sharing` (`engine_baselinenowcast()`) Whether to share estimates across the object's strata. `"none"` (default) fits every stratum independently. `"delay"` estimates the

delay PMF once on the pooled counts and applies it to each stratum; `"uncertainty"` shares the uncertainty parameters the same way; pass `c("delay", "uncertainty")` to share both. Passed straight to `baselinenowcast::baselinenowcast()`'s argument of the same name, and only meaningful when the object has strata.

`preprocess_args`, `expectation`, `reference`, `report`, `fit`

(`engine_epinowcast()`) `preprocess_args` is a list for `tbl_now_to_epinowcast()`, e.g. `list(max_delay = 30)`; the other four are `epinowcast::epinowcast()`'s module arguments. `epinowcast` **is unseeded unless you say so**: `enw_fit_opts()` has no `seed` argument of its own – `formals(epinowcast::enw_fit_opts)` on 0.7.0 lists `sampler`, `nowcast`, `pp`, `likelihood`, `likelihood_aggregation`, `threads_per_chain`, `debug`, `output_loglik`, `sparse_design`, `...` – but its `...` are forwarded to the `sampler` (`enw_sample()`, i.e. `cmdstanr::sample()`), which does. Pass `fit = epinowcast::enw_fit_opts(seed = 1)` and the seed rides through to the sampler; that is what makes a fit reproducible.

Two epinowcast 0.7.0 knobs worth knowing about, both reachable through the same pass-through:

- a **delay-only** fit – reporting-delay distribution conditional on per-reference-date totals, with the latent process disabled – via `obs = epinowcast::enw_obs(delay_only = TRUE, data = pobs)`. `obs` is not a named engine argument here, but `engine_epinowcast()` forwards `...` to `epinowcast::epinowcast()`, so passing it there works.
- a **structural** reporting effect (e.g. a fixed day-of-week reporting hazard) via `report = enw_report(structural = ...)`. Build the metadata with `enw_dayofweek_structural_reporting()`; this is separate from a temporal-effect covariate that lands on `metareport` and is referenced through `non_parametric =`.

`max_D`, `moving_window`, `specs`

(`engine_nobbs()`) Arguments of `NobBS::NobBS()` / `NobBS::NobBS.strat()`. `moving_window` counts **event periods and must not exceed the history you hand it** – ask for more and NobBS pads its grid backwards and returns zero for every date, with no error. `specs$quantiles` is filled from `quantile_levels` unless you set it.

`D`, `when`, `fit_method`, `control`

(`engine_surveillance()`) Arguments of `surveillance::nowcast()`. `fit_method` is that function's own `method` argument, renamed so it cannot collide with the engine's method. `when` defaults to `get_surveillance_when(x, length = D + 1)` and `control$dRange` to `get_surveillance_range()` – both built from the whole object, so every stratum is fitted on the same time axis.

`generation_time`, `delays`, `truncation`, `rt`, `obs`, `stan`, `convert_args`

(`engine_epinow2()`) Arguments of `EpiNow2::estimate_infections()` / `EpiNow2::regional_epinow` plus `convert_args` for `tbl_now_to_EpiNow2()`. **Read this before trusting the output**: EpiNow2 defaults to `delays = delay_opts()`, which is `Fixed(0)` – no reporting delay at all – and a one-day generation time. Those defaults describe a process with nothing to nowcast, so supply the epidemiology yourself. `truncation = trunc_opts()` is likewise `Fixed(0)` – **without a fitted truncation,** `estimate_infections()` **is a smooth through the incomplete recent days, not a nowcast**. The vignette's *EpiNow2* section walks through the two-step recipe (`estimate_truncation()` first, then pass its `get_parameters(...)[["truncation"]]` as `truncation` here).

Reproducibility. `EpiNow2::stan_opts()` picks a fresh random seed on every call (`seed = as.integer(runif(1, 1e8))`), so an unseeded fit cannot be reproduced – and a pathological sample cannot be told apart from a bad model afterwards. Pin it with `stan = stan_opts(samples = ..., warmup = ..., chains = ..., seed = <n>)`; `stan_opts()` forwards `seed` through to `rstan::sampling()` / `cmdstanr::sample()`.

Value

A `nowcast_engine`, as `engine()` returns.

See Also

`engine()`, `run_nowcast()`, `nowcast_backtest()`

Examples

```
engine_baselinenowcast(draws = 500)
engine_nobbs(max_D = 10, moving_window = 64)

# Fit epinowcast on the last 180 periods only; it scales with the number of
# reference dates, while the two engines below take the whole series.
engine_epinowcast(preprocess_args = list(max_delay = 30), min_date = 180)
engine_baselinenowcast()
engine_diseasenowcasting()
```

---

nowcast_ensemble *Combine several nowcasts into an ensemble*

Description

[Stable]

Takes the nowcasts produced by different modelling packages on the *same* `tbl_now` and combines them into a single tbl_nowcast. Ensembles are routinely better calibrated than any of their members, and because `run_nowcast()` puts every backend on the same tidy footing, combining them needs no reshaping on your side.

Usage

```
nowcast_ensemble(
  ...,
  type = c("quantile", "linear_pool"),
  weights = "equal",
  backtest = NULL,
  include_now = FALSE,
  quantile_levels = NULL,
  n_draws = 4000L,
  name = "ensemble",
  verbose = TRUE
)
```

Arguments

`...`	tbl_nowcast objects, or a single list of them. Named arguments rename the members.
`type`	How to combine the members: `"quantile"` **(default)** Average the members' predictive quantiles level by level (Vincentization). Applicable to every backend. `"linear_pool"` Pool the members' posterior draws into a mixture distribution and re-summarise it. Requires that every member returned draws, and generally yields wider intervals.
`weights`	Either the string `"equal"` (default), a numeric vector (named by method, or in the order the members were given), or one of `"inverse_score"` / `"optim"`. The last two require `backtest` and are passed to `nowcast_weights()`.
`backtest`	A `nowcast_backtest()` object, required when `weights` is `"inverse_score"` or `"optim"`.
`include_now`	Logical. When deriving performance weights from `backtest`, should rows at the nowcast members' own `now` dates be allowed into the weight-training window? Default `FALSE`; set `TRUE` only for an in-sample diagnostic.
`quantile_levels`	Quantile levels to report the ensemble at. Defaults to the levels shared by all members.
`n_draws`	Number of draws in the pooled sample when `type = "linear_pool"`. Default `4000`.
`name`	Name to record as the ensemble's `method`. Default `"ensemble"`.
`verbose`	Logical. Whether to report the weights that were used.

Details

An ensemble assumes that all members predict the same epidemiological quantity. The target dates, strata and quantile levels are checked here, but reporting-versus-revision semantics are currently a modelling convention: combine members that target the same quantity, and score the result with the matching `truth_axis` and `truth_type` in `score_nowcast()`.

Value

A tbl_nowcast whose `fit` property is the list of member nowcasts and whose `metadata` holds the `weights` and the combination `type`.

See Also

`run_nowcast()` to produce the nowcasts being combined; `nowcast_backtest()` and `nowcast_weights()` to decide how much to trust each one, instead of weighting them equally; `score_nowcast()` to check the ensemble beats its members. The *One call, many models* article builds one end to end.

Examples

```
toy <- function(method, shift) {
  predictions <- data.frame(
    onset_week = as.Date("2020-01-05"),
    .quantile_level = c(0.25, 0.5, 0.75),
    .value = c(8, 10, 13) + shift
  )
  tbl_nowcast(predictions = predictions, method = method, event_date = "onset_week")
}

nowcast_ensemble(toy("a", 0), toy("b", 4), verbose = FALSE)

# Unequal weights
nowcast_ensemble(toy("a", 0), toy("b", 4), weights = c(a = 0.75, b = 0.25), verbose = FALSE)
```

---

nowcast_fit.diseasenowcasting

Fit a nowcast with one modelling package

Description

[Stable]

`nowcast_fit()` and `nowcast_tidy()` are the two extension points of the nowcasting framework. Together they teach `run_nowcast()` about a new modelling package: `nowcast_fit()` runs the model, `nowcast_tidy()` turns whatever it returned into the tidy quantile format every other function in tbl.now understands.

Dispatch happens on the object built by `engine()`, so a method for `"mypackage"` is a function called `nowcast_fit.mypackage()`. It can live in any package.

Usage

```
## S3 method for class 'diseasenowcasting'
nowcast_fit(
  engine,
  x,
  ...,
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## S3 method for class 'baselinenowcast'
nowcast_fit(
  engine,
  x,
  ...,
  draws = 1000,
```

```
  delays_unit = NULL,
  max_delay = NULL,
  strata_sharing = "none",
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## S3 method for class 'epinowcast'
nowcast_fit(
  engine,
  x,
  ...,
  preprocess_args = list(),
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## S3 method for class 'NobBS'
nowcast_fit(
  engine,
  x,
  ...,
  specs = list(),
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## S3 method for class 'surveillance'
nowcast_fit(
  engine,
  x,
  ...,
  when = NULL,
  D = NULL,
  fit_method = "bayes.notrunc.bnb",
  control = list(),
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## S3 method for class 'EpiNow2'
nowcast_fit(
  engine,
  x,
  ...,
  convert_args = list(),
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
```

```
)

nowcast_fit(
  engine,
  x,
  ...,
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)

## Default S3 method:
nowcast_fit(
  engine,
  x,
  ...,
  quantile_levels = nowcast_quantile_levels(),
  verbose = TRUE
)
```

Arguments

engine	An engine() object – the modelling package plus its arguments. S3 dispatch is on its class, so a backend for "mypackage" is a function called nowcast_fit.mypackage(). The whole engine arrives, not just its name, so engine$args, engine$label and the rest are available to a backend that wants them.
x	A tbl_now object.
...	Arguments passed straight to the underlying modelling function. run_nowcast() splices the engine's own arguments in here.
quantile_levels	Numeric vector of probabilities. Most backends ignore it at fit time (the quantiles are computed from the draws afterwards), but some need to be told up front which levels to report.
verbose	Logical. Whether the backend (and the converters feeding it) should be chatty.
draws	("baselinenowcast" only) Number of nowcast samples to draw.
delays_unit	("baselinenowcast" only) Unit of the reporting triangle's delay axis; inferred from the object's time units when NULL.
max_delay	("baselinenowcast" only) Number of delay periods to keep, forwarded to tbl_now_to_baselinenowcast(). NULL keeps every delay, which a snapshot ("as of") series cannot be fitted with – see engine_baselinenowcast().
strata_sharing	("baselinenowcast" only) Whether to share estimates across strata: "none" (default) fits every stratum independently, "delay" pools the delay PMF, "uncertainty" pools the uncertainty parameters, or a vector combining them. Passed through to baselinenowcast::baselinenowcast()'s argument of the same name; only meaningful when the object has strata.
preprocess_args	("epinowcast" only) A list of arguments for tbl_now_to_epinowcast(), e.g. list(max_delay = 20).

specs ("NobBS" only) The specs list of NobBS::NobBS(). The quantiles element is filled from quantile_levels unless you set it.

when, D, fit_method, control
("surveillance" only) The when, D, method and control arguments of surveillance::nowcast(). when defaults to get_surveillance_when(x, length = D + 1), D to the largest delay in the data, and control$dRange to get_surveillance_range() – the grid running to get_now(), which a line list cannot express on its own. Both grids are built from the whole object, so every stratum is fitted on the same time axis. fit_method is **surveillance**'s method argument, renamed so it cannot collide with run_nowcast()'s own method.

convert_args ("EpiNow2" only) A list of arguments for tbl_now_to_EpiNow2(), e.g. list(accumulate = FALSE).

Value

nowcast_fit() returns the modelling package's own object, verbatim. It is stored in the fit property of the resulting tbl_nowcast, and it is the only thing nowcast_tidy() is given besides the tbl_now itself, so put whatever the tidying step will need into it.

See Also

nowcast_tidy(), run_nowcast(), list_nowcast_methods() and the *Adding your own nowcasting model* article for a worked example of a new backend.

Examples

```
# A minimal backend: two S3 methods and you are done.
nowcast_fit.constant <- function(engine, x, ..., quantile_levels, verbose = TRUE) {
  counts <- get_latest_reported_cases(x)
  list(dates = counts[[get_event_date(x)]], value = counts[[ncol(counts)]])
}

nowcast_tidy.constant <- function(engine, fit, x, ..., quantile_levels) {
  predictions <- tidyr::expand_grid(
    event_date = fit$dates, .quantile_level = quantile_levels
  )
  predictions$.value <- rep(fit$value, each = length(quantile_levels))
  names(predictions)[1] <- get_event_date(x)
  list(predictions = predictions, draws = NULL)
}
```

---

nowcast_quantile_levels

Default quantile levels for a nowcast

Description

[Stable]

The quantile levels `run_nowcast()` summarises a nowcast at by default: nine probabilities, symmetric about the median, spanning the 50%, 80%, 90% and 95% central intervals.

They are a **subset** of the 23 levels the US and European COVID-19 forecast hubs and FluSight ask for (0.01, 0.025, 0.05, then 0.10 to 0.90 in steps of 0.05, then 0.975 and 0.99). Nine cover the intervals people actually read at a fraction of the storage, and every one of them is a hub level, so the output still scores against hub submissions in **scoringutils** without interpolation. Pass `quantile_levels` explicitly when you need the full hub set:

```
hub_levels <- c(0.01, 0.025, seq(0.05, 0.95, by = 0.05), 0.975, 0.99)
run_nowcast(x, engine("baselinenowcast", quantile_levels = hub_levels))
```

The levels live on the `engine()`, not on `run_nowcast()`, because for some backends they are a fit-time model argument rather than a way of summarising afterwards.

Backends that expose draws can honour any levels you ask for. Ones that report a point estimate and a single interval (`"surveillance"`, `"EpiNow2"`) cannot, and say so rather than interpolating.

Usage

```
nowcast_quantile_levels()
```

Value

A numeric vector of nine probabilities in `(0, 1)`, sorted increasingly.

See Also

`engine()`, whose `quantile_levels` argument this is the default for; `run_nowcast()` and `nowcast_backtest()`, which report at these levels; `score_nowcast()` and `scoringutils::as_forecast_point()`, which score them.

Examples

```
nowcast_quantile_levels()

# The 50%, 80%, 90% and 95% central intervals, as lower/upper pairs.
matrix(nowcast_quantile_levels()[-5], ncol = 2)

# Ask an engine for something else -- here the full forecast-hub set.
hub_levels <- c(0.01, 0.025, seq(0.05, 0.95, by = 0.05), 0.975, 0.99)
engine("baselinenowcast", quantile_levels = hub_levels)
```

nowcast_summary_components *Individual blocks of a* tbl_now *summary*

Description

[Stable]

summary() answers a dozen questions about a tbl_now at once. When you only want one of them – for a report, a dashboard, or a check inside a script – call that block directly instead of computing the rest and filtering it away.

Every one of these returns the same schema as summary() itself, so they can be stacked with dplyr::bind_rows(), compared across datasets, or used alone.

- cases_per_date() – case counts per date on one axis.
- delay_summary() – the case-weighted delay distribution.
- zero_run_summary() – lengths of the runs of consecutive zero dates.
- prop_censored() – proportion of cases flagged censored.
- prop_revision_type() – proportion of cases per revision outcome.
- prop_strata() – proportion of cases per stratum.
- prop_covariate_levels() – proportion of cases per level of each categorical covariate.
- date_ranges() – totals, date ranges and now.
- triangle_occupancy() – how full the reporting triangle is, and how stale the object is.
- cumulative_growth() – ratio of one delay's running total to the previous one's.

Usage

```
cases_per_date(
  x,
  axis = c("event", "report", "revision"),
  by_strata = NULL,
  strata = NULL
)

delay_summary(
  x,
  delay = c("event_to_report", "event_to_revision", "report_to_revision"),
  by_strata = NULL,
  strata = NULL
)

zero_run_summary(
  x,
  axis = c("event", "report", "revision"),
  by_strata = NULL,
```

```
  strata = NULL
)

prop_censored(x, by_strata = NULL, strata = NULL)

prop_revision_type(x, by_strata = NULL, strata = NULL)

prop_strata(x, strata = NULL)

prop_covariate_levels(x, by_strata = NULL, strata = NULL)

date_ranges(x, by_strata = NULL, strata = NULL)

triangle_occupancy(x, by_strata = NULL, strata = NULL)

cumulative_growth(x, k = 7, by_strata = NULL, strata = NULL)
```

Arguments

x	A tbl_now object.
axis	Which time axis to describe: "event", "report" or "revision".
by_strata	Logical. Add one set of rows per stratum on top of the pooled ("all") rows. Defaults to TRUE when the object has strata.
strata	Character vector of columns to stratify by. Defaults to get_strata(x).
delay	Which delay to describe: "event_to_report" (the reporting delay), "event_to_revision" (the same span measured to the revision, so the two are comparable) or "report_to_revision" (the laboratory's turnaround, the .revision_delay column).
k	Number of delays for the growth ratios.

Value

A tibble in the schema documented in tbl_now_summary: one row per quantity and stratum, with component, quantity and stratum identifying the row and the remaining columns holding whichever statistics apply.

See Also

summary(), which stacks all of these into one table and documents the schema; diagnose() for what is *wrong* with the data rather than what is in it; autoplot() for the same information as pictures. The *Diagnosing a tbl_now* article walks through them in order.

Examples

```
data(denguedat)
# The last five years. The full twenty-year series gives the same shape
# of answer, it just takes longer to compute.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
ndata <- tbl_now(recent,
```

```
  event_date = "onset_week",
  report_date = "report_week",
  strata = "gender",
  verbose = FALSE
)

# How many cases per week of onset, and how long they took to be reported.
cases_per_date(ndata, axis = "event")
delay_summary(ndata)

# How sparse the series is.
zero_run_summary(ndata, axis = "event")

# What the data is made of, and how far it reaches.
prop_strata(ndata)
prop_censored(ndata)
date_ranges(ndata)
triangle_occupancy(ndata)

# How fast the running total is still growing. This is a distribution over
# event dates, so it fills `mean`/`q50` rather than the scalar `value`.
cumulative_growth(ndata, k = 3)

# Every block shares one schema, so they stack.
dplyr::bind_rows(
  date_ranges(ndata),
  delay_summary(ndata)
)
```

`nowcast_tidy.diseasenowcasting` *Standardise a fitted nowcast*

Description

[Stable]

The second extension point of the nowcasting framework (see `nowcast_fit()`). It receives the object the modelling package returned and must express its predictions in the tidy format tbl.now uses everywhere else.

Usage

```
## S3 method for class 'diseasenowcasting'
nowcast_tidy(engine, fit, x, ..., quantile_levels)

## S3 method for class 'baselinenowcast'
nowcast_tidy(engine, fit, x, ..., quantile_levels)
```

```
## S3 method for class 'epinowcast'
nowcast_tidy(engine, fit, x, ..., quantile_levels)

## S3 method for class 'NobBS'
nowcast_tidy(engine, fit, x, ..., quantile_levels)

## S3 method for class 'surveillance'
nowcast_tidy(engine, fit, x, ..., quantile_levels)

## S3 method for class 'EpiNow2'
nowcast_tidy(engine, fit, x, ..., quantile_levels)

nowcast_tidy(engine, fit, x, ..., quantile_levels)

## Default S3 method:
nowcast_tidy(engine, fit, x, ..., quantile_levels)
```

Arguments

engine	An engine() object; see nowcast_fit().
fit	The object returned by nowcast_fit().
x	The tbl_now the nowcast was produced from.
...	Not forwarded by run_nowcast(), which passes the user's ... to nowcast_fit() only. Anything the tidying step needs – a number of draws, a tuning parameter, a lookup table – must therefore travel inside the object nowcast_fit() returned.
quantile_levels	Numeric vector of probabilities the predictions should be summarised at.

Value

A list with two elements:

predictions A data frame with one row per (event date, stratum, quantile level) and the columns <event_date>, the strata columns, .quantile_level and .value.

draws Either NULL, or a data frame with one row per (event date, stratum, draw) and the columns <event_date>, the strata columns, .draw and .value.

When draws is supplied and predictions is NULL, run_nowcast() derives the quantiles from the draws for you.

See Also

nowcast_fit(), run_nowcast()

Examples

```
# See `?nowcast_fit` for a complete two-method backend.
methods(nowcast_tidy)
```

nowcast_weights *Ensemble weights from a backtest*

Description

[Stable]

Turns the retrospective scores of a `nowcast_backtest()` into a vector of weights for `nowcast_ensemble()`.

Usage

```
nowcast_weights(
  backtest,
  type = c("inverse_score", "optim", "equal"),
  now = NULL,
  include_now = FALSE,
  ...
)
```

Arguments

backtest	A nowcast_backtest object.
type	How to derive the weights: "inverse_score" **(default)** $w_i \propto 1/\overline{WIS}_i$. Cheap, robust, and never puts all the mass on one model. "optim" The weights on the simplex that minimise the WIS of the quantile-averaged ensemble over the training window. Better in principle, but prone to overfitting when the window is short. "equal" $w_i = 1/M$. Included so that the same code path can produce the unweighted ensemble.
now	Optional Date vector of nowcast origins to exclude from the weight-training window when include_now = FALSE. `nowcast_ensemble()` passes its members' own now values here when deriving performance weights.
include_now	Logical. Should rows at now be allowed into the weight-training window? Default FALSE; set TRUE for an in-sample diagnostic.
...	Unused.

Value

A named numeric vector of weights summing to 1.

See Also

`nowcast_backtest()`, which produces the scores these weights come from; `nowcast_ensemble()`, which consumes them; `engine()`'s label argument, which is what tells two configurations of the same package apart in the result.

Examples

```
data(denguedat)

# A short recent window keeps the example quick.
recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
dengue <- tbl_now(recent,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

## `example_engine()` is a toy that ignores the reporting delay entirely; it
# is used here only so the example runs without a modelling package.
## Swap in a real one -- `engine_baselinenowcast()`, `engine_epinowcast()`,
## `engine_nobbs()` -- for anything you intend to act on.

# Two engines that differ in how wide they claim their intervals are.
bt <- nowcast_backtest(dengue,
  example_engine(spread = 0.2, label = "narrow"),
  example_engine(spread = 0.5, label = "wide"),
  now_dates = as.Date(c("2010-10-04", "2010-11-15")),
  verbose = FALSE
)

# Weights sum to one, and the better-scoring engine takes the larger share.
nowcast_weights(bt)
sum(nowcast_weights(bt))

## Hand them to nowcast_ensemble() to pool the nowcasts they came from.
```

---

plot_cycles *Periodogram of the case counts or of the reporting delay*

Description

[Stable]

The `"seasonality"` / `"delay_seasonality"` panels of `autoplot()`, drawn on their own: a periodogram whose dominant peak is marked. For `type = "epidemic"` (green) the peak suggests a Fourier season length to pass to `temporal_effects()`; for `type = "report"` (red) it marks a cycle in the reporting delay itself, such as a weekly reporting rhythm.

Usage

```
plot_cycles(x, type = c("epidemic", "report", "revision"), ...)
```

Arguments

`x`	A `tbl_now()` object.
`type`	`"epidemic"` (default), `"report"` or `"revision"`.
`...`	Further arguments passed to `autoplot.tbl_now()`, e.g. `by_strata`, `strata`, `plotly` or `palette`.

Value

A **ggplot2** object (or a **plotly** widget when `plotly = TRUE`).

See Also

`autoplot.tbl_now()`, calendar_effect_plots.

Examples

```
data(denguedat)
dengue_now <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)
plot_cycles(dengue_now)
```

`plot_delay_distribution`

Empirical distribution of the reporting or revision delay

Description

[Stable]

The `"delay_distribution"` panel of `autoplot()`, drawn on its own: a case-count weighted histogram of `.delay`. For `count-cumulative` data it becomes the *cumulative growth by delay* instead — boxplots, on a log scale, of the ratio of each event date's cumulative count at a delay to its count at the previous delay.

`axis = "revision"` draws the same histogram of `.revision_delay`, the time from a report to its resolution, in the revision process's colours. A case still `"pending"` has no resolution, and so no revision delay, and does not appear.

Usage

```
plot_delay_distribution(
  x,
  axis = c("report", "revision"),
  by_revision_type = TRUE,
  ...
)
```

Arguments

`x`	A `tbl_now()` object.
`axis`	Which delay to draw: `"report"` (default), the time from the event to the report, or `"revision"`, the time from the report to its resolution. `"revision"` needs a revision process (see add_revision_date()).

by_revision_type
: Logical (default TRUE). Split the histogram by how each case eventually resolved — confirmed, pending, retracted and unknown, stacked, in the palette's outcome colours (see tbl_now_palette()). Whether a negative result comes back faster than a positive one is the question the split exists to answer, and diagnose_revision_delay() is the test of it. Ignored on an object with no revision axis, and when by_strata = TRUE, which already uses the fill for the strata.

...
: Further arguments passed to autoplot.tbl_now(), e.g. by_strata, strata, delay_distribution_xlim, plotly or palette.

Value

A **ggplot2** object (or a **plotly** widget when plotly = TRUE).

See Also

autoplot.tbl_now(), plot_delay_profiles(), plot_delay_drift(); diagnose_revision_delay() for the test behind the outcome split.

Examples

```
data(denguedat)
dengue_now <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)
plot_delay_distribution(dengue_now)

# On the revision axis, split by how each case resolved.
cases <- data.frame(
  onset = as.Date("2021-01-04") + rep(0:9, each = 4),
  visit = as.Date("2021-01-05") + rep(0:9, each = 4),
  result = as.Date("2021-01-05") + rep(0:9, each = 4) +
    rep(c(1, 1, 5, 6), times = 10),
  outcome = rep(c("confirmed", "confirmed", "retracted", "retracted"), times = 10)
)
flu <- tbl_now(cases,
  event_date = onset, report_date = visit,
  revision_date = result, revision_type = outcome,
  data_type = "linelist", verbose = FALSE
)
plot_delay_distribution(flu, axis = "revision")
```

---

plot_delay_drift *Visualise whether the reporting-delay distribution drifts over time*

Description

[Stable]

Draws a **rolling fan chart** of the count-weighted reporting-delay distribution indexed by **event date**: a solid line for the rolling median, a dashed line for the rolling mean, and two shaded bands (the 25-75% and 10-90% quantile ranges). Reading it left to right answers "do delays drift?" — a rising/falling centre line is *location* drift, widening/narrowing bands are *spread* drift.

Usage

```
plot_delay_drift(
  x,
  ...,
  window = NULL,
  step = NULL,
  min_n = 1,
  by_strata = FALSE,
  strata = NULL,
  changepoint = FALSE,
  level = 0.95,
  plotly = FALSE,
  axis = c("report", "revision"),
  linewidth = 1,
  grid_linewidth = 0.5,
  palette = .tbl_now_palette()
)
```

Arguments

x	A tbl_now object.
...	Unused.
window	Rolling-window width, in event-time **periods**. NULL (default) uses 7 periods regardless of the time unit — that is, 7 days for daily data and 7 weeks for weekly data.
step	Step between window centres, in periods. NULL (default) uses max(1, window / 4).
min_n	Minimum total case count for a window to be drawn (default 1).
by_strata	Logical (default FALSE). When TRUE, one fan is drawn per stratum (faceted).
strata	Character vector of columns to group on when by_strata = TRUE. NULL (default) uses the object's strata.
changepoint	Logical (default FALSE). When TRUE, mark the estimated abrupt change point of the **median** delay (Pettitt's test, on mature data) with a vertical line, when one is detected ($p < 0.05$). See diagnose_changepoint().
level	Completeness level for the immature-region shading (default 0.95; see autoplot()).
plotly	If TRUE, return an interactive **plotly** widget instead of a static plot. Default FALSE.

axis	Which time axis the delay is measured to: "report" (default) or "revision". Report-axis delays are measured from event to report; revision-axis delays are measured from report to revision, the same quantity as .revision_delay. Needs a revision process (see add_revision_date()); cases still "pending" are left out.
linewidth	Multiplier on the width of the mean and median delay lines. Default 1 (drawn at 0.6 and 0.8, so the median stays the heavier of the two at any setting).
grid_linewidth	Line width of the dashed maturity line and of the changepoint marker – the reference lines the package draws itself, not **ggplot2**'s panel grid. Default 0.5.
palette	A named colour palette (see tbl_now_palette()).

Details

Because recent event dates have not had time to be fully reported, their delay summaries are downward-biased (only short delays are observable yet). That immature region — event dates after the level incompleteness cutoff — is **shaded grey** and should not be read as drift. Pair the plot with diagnose_drift() for a formal test.

Value

A **ggplot2** object.

See Also

diagnose_drift() for the formal trend test behind this picture, and diagnose_changepoint() for an abrupt shift rather than a gradual one; plot_delay_distribution() for the delay pooled over the whole period; autoplot() and diagnostic_plot() for the galleries this belongs to.

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
plot_delay_drift(dengue)
```

---

plot_delay_profiles *Plot the per-date delay profiles*

Description

[Stable]

One translucent curve per date (see by) giving that date's share of reports at each delay, coloured by its mean delay. A batch is a lone right-shifted (long-delay) curve.

Usage

```
plot_delay_profiles(
  x,
  by = c("report", "event"),
  max_delay = NULL,
  plotly = FALSE,
  axis = c("report", "revision"),
  linewidth = 1,
  palette = .tbl_now_palette()
)
```

Arguments

x	A tbl_now() object.
by	One line per "report" date (default) or per "event" date.
max_delay	Largest delay to draw. NULL (default) caps at the delay covering 99% of reported mass.
plotly	If TRUE, return an interactive **plotly** widget instead of a static plot. Default FALSE.
axis	Which time axis the delay is measured on: "report" (default) or "revision". Report-axis delays are measured from event to report; revision-axis delays are measured from report to revision, the same quantity as .revision_delay. Needs a revision process (see add_revision_date()); cases still "pending" are left out.
linewidth	Multiplier on the width of the per-date curves. Default 1 (drawn at 0.4). The curves are deliberately faint and overplotted – it is their envelope that carries the message – so raising this on a long series fills the panel in.
palette	A named colour palette (see tbl_now_palette()).

Value

A **ggplot2** object (or a **plotly** widget when plotly = TRUE).

See Also

plot_delay_distribution() for the pooled delay distribution rather than one curve per date; plot_delay_drift() for whether those curves move over time; diagnose_batches2() for the test behind the eyeball; diagnostic_plot() for the whole gallery.

Examples

```
data(denguedat)
dn <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)
plot_delay_profiles(dn)
```

plot_epidemic_process *The epidemic process and the reporting process*

Description

[Stable]

The same cases, counted on two different clocks. Comparing the two is the single most useful thing you can do to tell a real outbreak from a reporting artifact.

- `plot_epidemic_process()` counts by **event date** – when the cases actually happened. Epidemics grow and shrink smoothly, so this curve should be smooth.
- `plot_reporting_process()` counts by **report date** – when news of them arrived. Reporting is administrative, so this curve is spiky: weekends, holidays and backlog releases all show up here.

A lone spike in the reporting process with nothing under it in the epidemic process is a **batch** – a day the system cleared its inbox, not a day people got sick. A spike in both is a genuine surge.

Usage

```
plot_reporting_process(
  x,
  plotly = FALSE,
  axis = c("report", "revision"),
  by_revision_type = TRUE,
  palette = .tbl_now_palette()
)

plot_epidemic_process(
  x,
  plotly = FALSE,
  axis = c("report", "revision"),
  palette = .tbl_now_palette()
)
```

Arguments

x	A `tbl_now()` object.
plotly	If `TRUE`, return an interactive **plotly** widget (hover, zoom) instead of a static **ggplot2** plot. Default `FALSE`.
axis	Which time axis to draw: `"report"` (default) or `"revision"`. On the revision axis the picture answers the laboratory's version of the question – when results arrived, rather than when reports did. Needs a revision process (see add_revision_date()); cases still `"pending"` have no revision date and are left out.

by_revision_type
: Logical (default TRUE), plot_reporting_process() only. Stack each bar by how the arrivals it counts eventually resolved – confirmed, pending, retracted and unknown, in the palette's outcome colours (see tbl_now_palette()) – so a day whose reports were mostly taken back is visible as such rather than as an ordinary day. Ignored on an object with no revision axis. It cannot be taken from count-cumulative data, which records running totals rather than cases, and warns and draws the unsplit bars there.

palette
: A named colour palette (see tbl_now_palette()). These two panels draw bars and nothing else, so they take no size or linewidth.

Details

Both are facetted by stratum when the object has strata.

Value

A **ggplot2** object (or a **plotly** widget when plotly = TRUE).

See Also

diagnostic_plot(), which draws these alongside the rest of the reporting-process gallery; plot_observed_cases() for the epidemic process with the incompleteness cutoff marked; diagnose_batches() to test a suspicious spike rather than eyeball it.

Examples

```
data(denguedat)
dn <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)

# When cases happened: smooth, because epidemics are.
plot_epidemic_process(dn)

# When news of them arrived: spikier, because reporting is administrative.
plot_reporting_process(dn)
```

---

plot_observed_cases *Observed epidemic process with the incompleteness line*

Description

[Stable]

The "epidemic" panel of autoplot(), drawn on its own: the latest reported counts per event_date, with a dashed vertical line marking where the data become incomplete (less than level of the delay distribution has arrived). Holidays from an attached temporal_effects() spec are marked with red dots.

plot_epidemic_process() draws the same curve without the incompleteness line, next to its reporting twin plot_reporting_process().

Usage

```
plot_observed_cases(x, ...)
```

Arguments

x	A tbl_now() object.
...	Further arguments passed to autoplot.tbl_now(), e.g. level, by_strata, strata, event_date_xlim, plotly or palette.

Value

A **ggplot2** object (or a **plotly** widget when plotly = TRUE).

See Also

autoplot.tbl_now(), plot_epidemic_process(), plot_reporting_process().

Examples

```
data(denguedat)
dengue_now <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)
plot_observed_cases(dengue_now)
```

---

plot_reporting_hexamap

Plot the reporting triangle as an age-period-cohort hexamap

Description

[Stable]

Draws the reporting triangle as a hexagonal age-period-cohort map, using the projection of Jalal and Burke (2020). Event date, report date and reporting delay are the cohort, period and age of the map (report = event + delay), and each (event, delay) cell is one point on the hexagonal lattice, coloured by its report count. Because a batch is a single **report date**, it appears as a clean **vertical stripe**; the fast-reporting bulk sits along the short-delay bottom edge.

Usage

```
plot_reporting_hexamap(
  x,
  max_delay = NULL,
  complete = FALSE,
  iso = NULL,
  iso_minor = NULL,
  format = "%d/%b/%y",
  max_cells = 12000L,
```

```
  trans = "sqrt",
  axis = c("report", "revision"),
  size = 1.5,
  shape = 16,
  text_size = 2.3,
  grid_linewidth_major = 0.3,
  grid_linewidth_minor = 0.15,
  axis_linewidth = 0.4,
  legend_width = 7,
  legend_height = 0.4,
  palette = .tbl_now_palette()
)
```

Arguments

x	A tbl_now() object.
max_delay	Largest delay (in report units) to draw. NULL (default) shows the observed range, auto-capped to respect max_cells.
complete	If TRUE, fill the whole observable triangle with zeros so a point is drawn for every observable cell. Default FALSE (observed cells only). Coerces linelist input to counts via to_count().
iso, iso_minor	Major and minor grid spacings (in arrival-axis units: report units on the report axis, revision units on the revision axis). NULL picks sensible defaults from the data.
format	Date format for the event/report tick labels (see strftime()). Default "%d/%b/%y".
max_cells	Safety cap on the number of points. Default 12000.
trans	Fill transform for the count scale. Default "sqrt".
axis	Which time axis to draw: "report" (default) or "revision". On the revision axis the picture answers the laboratory's version of the question – when results arrived, rather than when reports did. Needs a revision process (see add_revision_date()); cases still "pending" have no revision date and are left out.
size	Size of the plotted points, in millimetres, as **ggplot2** measures it. Default 1.5. See **Details** for why there is no data-dependent default.
shape	Point shape, passed to ggplot2::geom_point(). Default 16 (a solid circle); 15 gives squares, which tile the lattice more closely. The count is mapped to colour, so use a solid shape (0-20) – the fillable shapes 21-25 would draw the count on the border only.
text_size	Size of the event-, report- and delay-axis tick labels. Default 2.3. The axis *titles* scale with it.
grid_linewidth_major, grid_linewidth_minor	Line widths of the major and minor triangular grids this function draws (iso and iso_minor spacing). These are the package's own grids, not **ggplot2**'s – the panel grid is switched off here. Defaults 0.3 and 0.15.
axis_linewidth	Line width of the delay-axis spine and its ticks. Default 0.4.

`legend_width`, `legend_height`
: Size of the count colourbar, as unit objects or as numbers in centimetres. Defaults 7 and `0.4` cm.

`palette`
: A named colour palette (see `tbl_now_palette()`).

Details

The three axes are read off three families of iso-lines: **report date** (period) runs vertically, **delay** (age) up the right-hand spine, and **event date** (cohort) up the left. A major/minor triangular grid is drawn so any point can be traced back to its event date, report date and delay.

The number of points is `#\{observed (event, delay) cells\}`, which grows with the delay range. To stay responsive the delay axis is capped so at most `max_cells` points are drawn (raise `max_cells`, or set `max_delay`, to change this). `complete = TRUE` first fills the whole observable triangle with explicit zeros (via `complete_zeroes()`) so the empty cells are shown too.

A point is sized in millimetres and the lattice is sized in data units, so no default `size` can be right for every combination of cell count and figure size – which is exactly why `size` exists. Raise it until the points nearly touch for the figure you are actually drawing.

Value

A **ggplot2** object.

References

Jalal, H. and Burke, D. S. (2020). Hexamaps for Age-Period-Cohort Data Visualization. *Epidemiology* **31**, e47-e49.

See Also

`plot_reporting_triangle()` for the same data on ordinary axes, where the third quantity has to be read off the diagonals; `diagnostic_plot()` for the whole gallery.

Examples

```
data(denguedat)
dn <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)
plot_reporting_hexamap(dn)
```

`plot_reporting_triangle` *Plot the reporting triangle*

Description

[Stable]

Tiles over (event date, delay), filled by the reported count. Cells that are **observable but empty** (a genuine reported zero) are drawn in a muted blue; cells that are **not yet reportable** (report date beyond `now`, the upper-right wedge) are left blank. A **third axis for report date** is drawn as evenly spaced dashed diagonals (`report = event + delay`) running up-right at 45 degrees, so all three quantities – event date, delay and report date – can be read off one plot. A batch is a single report date, i.e. one such diagonal.

Usage

```
plot_reporting_triangle(
  x,
  max_delay = NULL,
  report_ticks = 6L,
  mark_batches = 0L,
  plotly = FALSE,
  axis = c("report", "revision"),
  size = 1,
  grid_linewidth = 0.3,
  palette = .tbl_now_palette()
)
```

Arguments

`x`	A `tbl_now()` object.
`max_delay`	Largest delay to draw. `NULL` (default) caps at the delay covering 99% of reported mass.
`report_ticks`	Integer: how many evenly spaced report-date diagonals to draw as the third (report-date) axis. `0` disables it. Default `6`.
`mark_batches`	Integer: additionally highlight this many of the biggest batch stripes with a stronger dashed diagonal labelled by report date. `0` (default) disables it. Found cheaply from volume spikes, not `diagnose_batches()`.
`plotly`	If `TRUE`, return an interactive **plotly** widget instead of a static plot. Default `FALSE`.
`axis`	Which time axis to draw: `"report"` (default) or `"revision"`. On the revision axis the picture answers the laboratory's version of the question – when results arrived, rather than when reports did. Needs a revision process (see `add_revision_date()`); cases still `"pending"` have no revision date and are left out.
`size`	Multiplier on the size of the report-date and batch-stripe labels. Default `1`.
`grid_linewidth`	Line width of the iso-report diagonals this function draws as the third axis – the package's own grid, not **ggplot2**'s. Default `0.3`; the `mark_batches` stripes are drawn a third heavier.
`palette`	A named colour palette (see `tbl_now_palette()`).

Value

A **ggplot2** object (or a **plotly** widget when `plotly = TRUE`).

See Also

`plot_reporting_hexamap()` for the same grid drawn so that event date, report date and delay are all read the same way; `plot_delay_profiles()` for one curve per date instead of a grid; `complete_zeroes()` to fill the cells that are genuinely zero; `diagnostic_plot()` for the whole gallery.

Examples

```
data(denguedat)
dn <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)

# Rows are event dates, columns are delays. The blank upper-right wedge is
# the future: those reports cannot have arrived yet. That wedge is what a
# nowcast fills in.
plot_reporting_triangle(dn)
```

---

plot_revision_status *How much of each day has been resolved*

Description

[Stable]

The share of each event date's cases that are **confirmed**, **retracted** or still **pending**, as of the object's `now`.

This is the picture of the *resolution front*. The oldest dates are almost entirely resolved; the most recent ones are mostly pending, because the laboratory has not caught up yet. Where that front sits tells you how far back the confirmed counts can be trusted – and a day that is 80% pending is a day whose confirmed count means very little.

Usage

```
plot_revision_status(
  x,
  by = NULL,
  proportion = TRUE,
  palette = .tbl_now_palette()
)
```

Arguments

x	A tbl_now with a revision process.
by	Optional stratum column to facet by.
proportion	When TRUE (default) the bands are shares summing to 1; FALSE shows the counts instead, which keeps the epidemic curve visible.
palette	A named colour palette (see tbl_now_palette()). The plot is drawn entirely with stacked areas, so it takes no size or linewidth.

Value

A ggplot.

Reading it

The pending band widening towards the right is normal and expected – it is the same right-truncation a nowcast exists to correct, one axis over. What is *not* normal is a pending band that stays wide far from the now: those cases were reported and then never resolved, and they will never be. Consider censor_revision_delays_above().

A retracted share that changes over time is worth investigating: it usually means the testing criteria or the case definition changed, not that the disease did.

Colours

confirmed is drawn with the palette's epidemic role (it is a real case – the epidemic process), retracted with reporting (it was removed by the reporting process), and pending with the neutral pending role (not yet known either way). Override any of them through palette.

See Also

diagnose_revision_delay(), revised_cases.

Examples

```
cases <- data.frame(
  onset = as.Date("2021-01-04") + rep(0:9, each = 4),
  visit = as.Date("2021-01-05") + rep(0:9, each = 4),
  result = as.Date("2021-01-07") + rep(0:9, each = 4),
  outcome = rep(c("confirmed", "confirmed", "retracted", "pending"), times = 10)
)
cases$result[cases$outcome == "pending"] <- as.Date(NA)
flu <- tbl_now(cases,
  event_date = onset, report_date = visit,
  revision_date = result, revision_type = outcome,
  data_type = "linelist", verbose = FALSE
)

plot_revision_status(flu)
```

`plot_transport_discriminant` *Plot the transport-discriminant plane*

Description

[Stable]

Places each report date by its creation score (x) and transport / deficit score (y) from `transport_discriminant()`, shading the region that decides the batch call. Surges are not distinguished here (they fold into the quiet background) since only the batch call is of interest.

Usage

```
plot_transport_discriminant(
  x,
  ...,
  plotly = FALSE,
  size = 1,
  grid_linewidth = 0.3,
  palette = .tbl_now_palette()
)
```

Arguments

`x`	A `tbl_now()` object.
`...`	Passed to `transport_discriminant()` (e.g. `lookback`, `period`, `alpha`).
`plotly`	If `TRUE`, return an interactive **plotly** widget instead of a static plot. Default `FALSE`.
`size`	Multiplier on the size of the points and their date labels. Default `1`: unflagged points are drawn at `1.1`, confirmed batches at `2.6`. It is a multiplier rather than an absolute size precisely so that enlarging the marks keeps the flagged ones bigger than the rest.
`grid_linewidth`	Line width of the zero lines and the dashed significance thresholds this function draws – the package's own reference grid, not **ggplot2**'s. Default `0.3`.
`palette`	A named colour palette (see `tbl_now_palette()`).

Details

Only the `diagnose_batches()`-confirmed batches (Benjamini-Hochberg-corrected) are coloured red; the dashed lines and shaded region are a reference for where a batch sits (deficit cleared, and significant), not the flagging rule. The most extreme-looking points (far left, far up) are *holds* – windows still depleted because the release has not happened yet – not batches. A genuine batch sits in the band just to the right of the vertical line, once the window total recovers.

Value

A **ggplot2** object (or a **plotly** widget when `plotly = TRUE`).

See Also

`transport_discriminant()` for the numbers behind the plane; `diagnose_batches()` for the hypothesis test that flags the red points; plot_reporting_process() for the series they come from; `diagnostic_plot()` for the whole gallery.

Examples

```
data(denguedat)
# The two and a half years around the 1996 and 1997 backlog dumps, so that
# the plane has red points on it without scanning the whole series.
window <- denguedat[
  denguedat$onset_week >= as.Date("1995-06-01") &
    denguedat$onset_week <= as.Date("1998-01-01"),
]
dn <- tbl_now(window, onset_week, report_week, verbose = FALSE)
plot_transport_discriminant(dn)
```

`print.tbl_now_diagnosis` *Print a* `tbl_now` *diagnosis*

Description

[Stable]

Prints the findings `diagnose()` returned as a report: the errors, warnings and notes in full, each with its hint, and the checks that passed and that could not be assessed as one line each.

The object is an ordinary tibble underneath, so `print(tibble::as_tibble(x))` gives the table and every `dplyr` verb still works on it.

Usage

```
## S3 method for class 'tbl_now_diagnosis'
print(x, ..., all = FALSE)
```

Arguments

`x`	A findings tibble, from `diagnose()` or one of the nowcast_diagnose_components.
`...`	Unused.
`all`	Logical. Spell out the `ok` and `skipped` findings too, instead of counting them. Defaults to `FALSE`, and to `TRUE` when there is nothing else to report – a block that found nothing wrong would otherwise print an empty report.

Value

x, invisibly.

See Also

diagnose(), nowcast_diagnose_components

Examples

```
data(denguedat)
# The last five years. The full twenty-year series gives the same shape
# of answer, it just takes longer to compute.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
ndata <- tbl_now(recent,
  event_date = "onset_week", report_date = "report_week",
  strata = "gender", verbose = FALSE
)

diagnose(ndata)

# Every finding, including the ones that passed.
print(diagnose(ndata), all = TRUE)

# Still a tibble.
print(tibble::as_tibble(diagnose(ndata)))
```

---

print.tbl_now_summary_table

Print a tbl_now *summary*

Description

[Stable]

Prints the table summary() returned one **component** at a time, dropping the columns that component does not populate. The full schema is wide because it has to hold every block's statistics at once; no single block fills more than a handful of them, and a table that is mostly NA is hard to read for a reason that has nothing to do with the data.

The object is an ordinary tibble underneath, so `print(tibble::as_tibble(x))` gives the whole schema back and every dplyr verb still works on it.

Usage

```
## S3 method for class 'tbl_now_summary_table'
print(x, ..., n = 10)
```

Arguments

x	A summary tibble, from summary() or one of the nowcast_summary_components.
...	Unused.
n	Maximum number of rows to show per component. Inf shows all of them.

Value

x, invisibly.

See Also

summary(), nowcast_summary_components

Examples

```
data(denguedat)
# The last five years. The full twenty-year series gives the same shape
# of answer, it just takes longer to compute.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
ndata <- tbl_now(recent,
  event_date = "onset_week", report_date = "report_week",
  strata = "gender", verbose = FALSE
)

summary(ndata)

# One block on its own prints the same way.
delay_summary(ndata)

# Still a tibble.
print(tibble::as_tibble(summary(ndata)))
```

---

revised_cases *Cases at a chosen point in the revision process*

Description

[Stable]

The same three questions as get_latest_reported_cases(), asked of the **third** date: not when the system heard about a case, but when the laboratory settled it.

- get_initial_revised_cases() – the count as of the **first** result to come back for that event date.
- get_latest_revised_cases() – the count as of the **most recent** result: everything settled so far.
- get_nth_revised_cases() – the count settled **within a given delay of the event**.

A case that is still "pending" has no revision date, so it has not arrived on this axis and none of these count it. That is the point: the gap between get_latest_reported_cases() and get_latest_revised_cases() is the backlog the laboratory still owes you.

Usage

```
get_latest_revised_cases(x, type = "total")

get_initial_revised_cases(x, type = "total")

get_nth_revised_cases(x, delay, type = "total")
```

Arguments

x	A tbl_now with a revision process (see add_revision_date()).
type	Which cases to count. One of: "total" (default) every case, whatever the outcome. On the revision axis that means every case that has been settled at all. "confirmed", "retracted", "pending" only the cases with that outcome. "pending" is a reporting-axis question only – a pending case has no revision date – and the revision getters refuse it. "unknown" the cases whose revision_type is NA: settled, but the data does not say which way. "net" confirmed **minus** retracted – the running total as a surveillance system publishes it, which can go **down** when a case is withdrawn. This is the quantity a count-cumulative stream actually reports, and the one **diseasenowcasting**'s cumulative signed-change likelihood is built for. "by_type" one row per outcome instead of one number: the outcome column joins the keys, so you get pending, confirmed and retracted side by side. On an object with no revision process anything but "total" warns and pools, because there is no outcome to filter on.
delay	A single non-negative number (or Inf) giving the longest delay **from the event** to include, in the object's units. Only used by get_nth_revised_cases().

Value

A count-cumulative tbl_now with one row per event date (and stratum, grouping column, and outcome when type = "by_type"), carrying the event, report and revision dates of the selected arrival, the generated numeric columns, and the count.

Which date the count is indexed by

By the **event date**, as every other get_*_cases() function is. A case confirmed three weeks after onset still belongs to the week it began. If you want counts by revision date instead, group on get_revision_date(x) yourself – that is a different question (how busy was the laboratory) and this package does not silently answer it.

Which delay `get_nth_revised_cases()` counts

The delay **from the event**, so that `get_nth_reported_cases(x, 7)` and `get_nth_revised_cases(x, 7)` describe the same seven days and can be read against each other. It is deliberately *not* `.revision_delay`, which is the laboratory's turnaround measured from the report. `diagnose_drift()` and summary() make the same choice for the same reason.

Grouping is respected

As with the reporting-axis getters: the caller's grouping becomes a key and comes back on the result. See get_latest_reported_cases().

See Also

get_latest_reported_cases() for the same counts on the reporting process; add_revision_date() to attach a revision; revision_delay for how long resolution takes; `plot_revision_status()` to see confirmed, retracted and pending over time.

Examples

```
cases <- data.frame(
  onset = as.Date("2021-01-04") + c(0, 0, 1, 1, 2),
  visit = as.Date("2021-01-05") + c(0, 0, 1, 1, 2),
  result = as.Date("2021-01-06") + c(0, 0, 1, 1, 2),
  outcome = c("confirmed", "retracted", "confirmed", "confirmed", "retracted")
)
flu <- tbl_now(cases,
  event_date = onset, report_date = visit,
  revision_date = result, revision_type = outcome,
  data_type = "linelist", verbose = FALSE
)

# Three answers to "how many cases were there?".
get_latest_reported_cases(flu) # everything reported
get_latest_revised_cases(flu, type = "confirmed") # only the positives
get_latest_revised_cases(flu, type = "net") # positives minus withdrawals

# Every outcome side by side.
get_latest_revised_cases(flu, type = "by_type")

# And the same question asked earlier in the process: what had come back by
# the first result, and within two days of onset.
get_initial_revised_cases(flu)
get_nth_revised_cases(flu, delay = 2)
```

revision_delay *Compare revision delays between confirmed and retracted cases*

Description

[Stable]

A negative result often comes back faster than a positive one – or slower, if positives are prioritised. Either way the delay from report to resolution is **not** the same distribution for the two outcomes, and a nowcast that assumes it is will be wrong about how many pending cases are still to be confirmed.

`diagnose_revision_delay()` compares the two delay distributions; `plot_delay_distribution()` with `axis = "revision"` shows them.

Usage

```
diagnose_revision_delay(x, by = NULL)
```

Arguments

`x`	A `tbl_now` with a revision process.
`by`	Optional stratum column to compare within; `NULL` (default) pools.

Value

`diagnose_revision_delay()` returns a one-row-per-comparison `tibble` with `stratum`, `n_confirmed`, `n_retracted`, `median_confirmed`, `median_retracted`, `difference`, `statistic` and `p.value`.

The test

A two-sided **Wilcoxon rank-sum** test on the revision delays. It is used rather than a t-test because reporting delays are strongly right-skewed and frequently have a point mass at zero, so a difference in means is neither robust nor the quantity of interest – what matters is whether one outcome resolves systematically sooner.

A small p-value says the two delay distributions differ. It does **not** say the difference matters: with tens of thousands of records a one-hour difference is significant and irrelevant, so read `difference` (the gap in median days) alongside it.

Rows with a missing or negative delay are dropped, and how many is reported in the `dropped` attribute of the result. A negative revision delay means the record is revised before it was reported, which the timeline forbids.

See Also

add_revision_date() to attach a revision process; censor_revision_delays_above() for resolutions that never arrive; revised_cases for counting the outcomes; `plot_delay_distribution()` with `axis = "revision"` for the picture of the same comparison; `diagnose_drift()` for the same question about the *reporting* delay over time. The *Diagnosing a tbl_now* article puts this alongside the other checks.

Examples

```
cases <- data.frame(
  onset = as.Date("2021-01-04") + rep(0:9, each = 4),
  visit = as.Date("2021-01-05") + rep(0:9, each = 4),
  result = as.Date("2021-01-05") + rep(0:9, each = 4) +
    rep(c(1, 1, 5, 6), times = 10),
  outcome = rep(c("confirmed", "confirmed", "retracted", "retracted"), times = 10)
)
flu <- tbl_now(cases,
  event_date = onset, report_date = visit,
  revision_date = result, revision_type = outcome,
  data_type = "linelist", verbose = FALSE
)

# Retractions here come back about four days later than revisions, and
# the test says so.
diagnose_revision_delay(flu)

# The same comparison as a picture.
plot_delay_distribution(flu, axis = "revision")
```

---

run_nowcast *Nowcast a* tbl_now *with any supported modelling package*

Description

[Stable]

Fits a nowcasting model to a `tbl_now` and returns the result in a package-agnostic shape, so that models from different packages can be compared, scored and combined (`nowcast_ensemble()`) without any manual reshaping.

The function is named `run_nowcast()` rather than `nowcast()` because **diseasenowcasting** already exports a `nowcast()` function; keeping the names distinct means both packages can be attached at once.

Usage

```
run_nowcast(x, engine = engine_diseasenowcasting(), verbose = TRUE)
```

Arguments

`x`	A `tbl_now` object.
`engine`	A `engine()` object: the modelling package **and every argument it takes**, including `min_date` and `quantile_levels`. Defaults to `engine_diseasenowcasting()`. The data and `verbose` are the only things that sit outside it. That is the point: an argument in an outer `...` had to be routed to the right backend by name, and one that missed simply vanished, leaving the model at its default with nothing to say so.
`verbose`	Logical. Whether to report what is being done.

Value

A tbl_nowcast object.

Engines

`engine_diseasenowcasting()` Bayesian structural time series. The `tbl_now` is passed in directly, so strata and temporal effects are picked up automatically.

`engine_baselinenowcast()` Fast, assumption-light baseline built from the reporting triangle. Stratified objects are nowcast one triangle per stratum.

`engine_epinowcast()` Bayesian model with separate delay and reference modules; `preprocess_args` controls `tbl_now_to_epinowcast()`.

`engine_surveillance()` Höhle & an der Heiden's nowcast, fed by `tbl_now_to_surveillance()`. The package models one series, so a stratified object is fitted one stratum at a time.

`engine_epinow2()` `EpiNow2::estimate_infections()`, fed by `tbl_now_to_EpiNow2()`; `EpiNow2::regional_epinow(` when the object declares strata.

`engine_nobbs()` Nowcasting by Bayesian Smoothing, fed by `tbl_now_to_nobbs()`, which expands counts to the one row per case **NobBS** counts.

`engine()` covers any other registered method, including one you wrote yourself. Every modelling package is an optional dependency: it is only needed when you ask for its engine.

What the object contributes, and what it does not

`run_nowcast()` reads the `tbl_now`'s declarations and hands each package the shape it wants. Three of those declarations behave differently enough to be worth stating plainly.

Strata. How many strata a backend can honour is a property of the *package*, not of this one. Where a backend cannot, it warns and pools rather than pretending:

engine	how strata are modelled
`"baselinenowcast"`	one reporting triangle, and one fit, per stratum
`"surveillance"`	one fit per stratum; the package models a single series
`"EpiNow2"`	`regional_epinow()` instead of `estimate_infections()`
`"epinowcast"`	passed to the model as `by`, so they are fitted jointly
`"diseasenowcasting"`	fitted jointly; the package returns a `[draws x time x stratum]` array, one slice per combination
`"NobBS"`	`NobBS.strat()` instead of `NobBS()`

Every backend takes any number of strata. The two that model one series at a time (`"surveillance"`) or accept a single column (`"NobBS.strat()"`) are given the *interaction* of the declared columns, which is what nowcasting each combination separately means; the label is split back into its columns on the way out.

The one thing that can go wrong is a stratum **value** that already contains the `" | "` used to join them. That is an error rather than a guess, because silently mis-assigning strata is worse than refusing.

Whatever strata columns come back are what the result reports as its `strata`.

Columns you did **not** declare are summed away by the converters (see `tbl_now_to_baselinenowcast()`). A `tbl_now` built from `covid_colombia` without `strata = sex` is nowcast as one pooled series, not silently split.

Temporal effects. add_temporal_effects() specs are lazy; the converters materialise them into ordinary columns, so they travel with the data. Whether the *model* then uses them is a separate question, and mostly the answer is "only if you say so":

- "diseasenowcasting" receives the tbl_now itself and reads the effects off it, so they enter the model with no further work.
- "epinowcast" carries them as covariates you name in a module formula, e.g. reference = epinowcast::enw_reference(~ 1 + day_of_week, data = ...). Without a formula referring to them they are inert.
- every other backend ignores them: the columns ride along so you can split on them, and nothing else happens.

Censored delays. A per-case censoring flag (see add_is_censored_report()) puts a censored and an uncensored row in the same (event date, report date) cell, and a reporting triangle has one slot per cell. Every backend that goes through a converter therefore **collapses the flag with a warning** — counts are summed over it, line lists drop the column. "diseasenowcasting" is the exception: it is handed the object untouched, so the flag reaches the package intact. To *estimate* a delay distribution from censored data, use tbl_now_to_epidist() instead; nowcasting and delay estimation are different jobs.

How each model is specified, and how to change it

run_nowcast() does not invent priors or model structure: it calls each package with **that package's own defaults** and passes the engine's arguments straight through. The defaults are not always the ones you want, and two are worth knowing before you read the output.

engine_epinowcast() runs three modules, all at their package defaults. The expectation module is enw_expectation(r = ~ 0 + (1 | day:.group)) – a **random effect per day** on the growth rate, which is a random walk on the log expected counts in all but name. The reference module is enw_reference(parametric = ~ 1, distribution = "lognormal") – a **single lognormal reporting delay, constant over time**. The report module is enw_report(non_parametric = ~ 0) – **no day-of-week reporting effect**. Each is a named argument of the engine, and preprocess_args carries tbl_now_to_epinowcast()'s own arguments:

```
run_nowcast(nowobj, engine_epinowcast(
  preprocess_args = list(max_delay = 30),
  report = epinowcast::enw_report(~ 1 + day_of_week, data = pobs),
  fit    = epinowcast::enw_fit_opts(chains = 4, iter_sampling = 1000)
))
```

engine_epinow2() – read this before trusting the output. EpiNow2::estimate_infections() defaults to delays = delay_opts(), which is Fixed(0): **no reporting delay at all**. Its generation_time = gt_opts() is Fixed(1), a one-day generation time. Those defaults describe a process with nothing to nowcast, so supply the epidemiology yourself. EpiNow2 also models R_t with a **Gaussian process** by default (rt_opts(rw = 0, gp_on = "R_t-1")) rather than a random walk:

```
run_nowcast(nowobj, engine_epinow2(
  generation_time = EpiNow2::gt_opts(EpiNow2::example_generation_time),
  delays          = EpiNow2::delay_opts(EpiNow2::example_reporting_delay),
  rt              = EpiNow2::rt_opts(rw = 7)   # weekly random walk instead
))
```

`engine_diseasenowcasting()` uses the package's own defaults, reading strata, covariates and temporal effects off the object. `model`, `type` and `n_draws` go to `diseasenowcasting::nowcast()`.

`engine_baselinenowcast()` is not Bayesian and has no priors: the delay is estimated from the reporting triangle and applied. `draws` sets the number of nowcast samples, and `max_delay` caps the triangle's width.

`engine_nobbs()` has a fixed model; what you tune is `max_D` (maximum delay) and `moving_window` (how much history is fitted). `moving_window` counts **event periods and must not exceed the history you hand it** – ask for more and NobBS pads its grid backwards and returns zero for every date, with no error. `specs` takes its prior list.

`engine_surveillance()` takes `fit_method`, which is **surveillance**'s own `method` argument renamed so it cannot collide with the engine's method; it defaults to `"bayes.notrunc.bnb"`. `D`, `when` and `control` are derived from the object when you do not give them.

See Also

The *Get started* vignette for where a fit sits in the workflow; `engine()` and nowcast_engines to specify which model to fit and how; autoplot() and tidy() to look at the result; `nowcast_ensemble()` to combine several nowcasts; `nowcast_backtest()` and `score_nowcast()` to find out whether they are any good; `nowcast_fit()` and `nowcast_tidy()` to add a backend of your own, and `example_engine()` for the shortest complete one. The *One dataset, many nowcasts* article fits the same data with every supported package.

Examples

```
data(denguedat)

# A short recent window keeps the example quick.
recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
dengue <- tbl_now(recent,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

# Every nowcast goes the same way: describe the model with an engine, then
## hand it and the data to `run_nowcast()`.
#
## `example_engine()` is a toy that ignores the reporting delay entirely; it is
# used here only so the example runs without a modelling package. Swap in a
## real one -- `engine_baselinenowcast()`, `engine_epinowcast()`,
## `engine_nobbs()` -- for anything you intend to act on.
nc <- run_nowcast(dengue, example_engine(), verbose = FALSE)
nc

# The result is a tbl_nowcast: one row per event date and quantile level.
head(tibble::as_tibble(nc))

# A real model is the same call with a real engine.
if (requireNamespace("baselinenowcast", quietly = TRUE)) {
  nc <- run_nowcast(dengue, engine_baselinenowcast(draws = 100), verbose = FALSE)
  nc
}
```

sari_bh *sari_bh: Severe Acute Respiratory Illness data from Belo Horizonte (Brazil), 2020-2022*

Description

An anonymized dataset containing cases of Severe Acute Respiratory Illness (SARI) cases from the Brazilian municipality, Belo Horizonte, with symptom onset varying from 2019-12-29 to 2022-03-27.

Usage

```
sari_bh
```

Format

A data frame with 65404 rows and 5 variables:

symptom_onset_date Date of onset symptoms, formatted in ISO8601 as YYYY-MM-DD.

record_date Date the case was recorded in the surveillance system, formatted in ISO8601 as YYYY-MM-DD. May be `NA` when no record date was available.

final_classification Final classification of the case: one of `"Influenza"`, `"Other respiratory virus"`, `"Other ethiological agent"`, `"Not specified"`, or `"COVID-19"`.

case_evolution Case evolution outcome: one of `"Cured"`, `"Dead"`, `"Dead (other causes)"`, `"Unknown"`, or `"Not in dictionary"`.

age_yrs Age of the patient in years.

Source

Redistributed from the **nowcaster** package (https://github.com/covid19br/nowcaster), which derives it from the Brazilian Ministry of Health's SIVEP-Gripe respiratory surveillance system, published on the OpenDataSUS portal.

References

From the `nowcaster` package. Bastos, S L, Economou, Theodoros, Gomes, FC M, Villela, AM D, Coelho, C F, Cruz, G O, Stoner, Oliver, Bailey, Trevor, Codeço, T C (2019). "A Modelling approach for correcting reporting delays in disease surveillance data." *Statistics in medicine*, *38*(22), 4363-4377. doi:10.1002/sim.8303

score_nowcast *Score a nowcast against observed data*

Description

[Stable]

A nowcast is a claim about numbers that are not in yet. Once the late reports arrive you can ask how good the claim was.

- `score_nowcast()` scores it here: the **weighted interval score** (WIS, lower is better), the absolute error of the median, and whether the truth fell inside the 50% and 90% intervals – one row per event date and stratum.
- `scoringutils::as_forecast_point()` hands the median prediction and the same truth to **scoringutils**, so you can use its point-score functions and plots.
- `scoringutils::as_forecast_quantile()` and `scoringutils::as_forecast_sample()` accept the same objects directly.

All three are **scoringutils** generics; this package only supplies the methods, so call them qualified (or after `library(scoringutils)`).

In each case `truth` is a `tbl_now` seen *later*, after the information the nowcast was predicting has arrived. The observed counts are computed from `truth_axis` and `truth_type`: by default this is get_latest_reported_cases() with `type = "total"`, while `truth_axis = "revision"` uses `get_latest_revised_cases()`. There is no observed column to name; the count column is read from the `tbl_now`.

Usage

```
score_nowcast(
  x,
  truth = NULL,
  truth_axis = c("report", "revision"),
  truth_type = "total"
)

## S3 method for class 'nowcast_backtest'
as_forecast_quantile(
  data,
  ...,
  truth = NULL,
  truth_axis = c("report", "revision"),
  truth_type = "total"
)

## S3 method for class 'nowcast_backtest'
as_forecast_point(
  data,
```

```
  ...,
  truth = NULL,
  truth_axis = c("report", "revision"),
  truth_type = "total"
)

## S3 method for class 'nowcast_backtest'
as_forecast_sample(
  data,
  ...,
  truth = NULL,
  truth_axis = c("report", "revision"),
  truth_type = "total"
)
```

Arguments

x	A tbl_nowcast.
truth	The tbl_now the nowcast is scored against – normally the *full* object, still holding the reports or revisions that arrived after the nowcast's now. Its observed counts per event date are worked out from truth_axis and truth_type, aggregated over anything that is not a stratum, with the count column read off the object (get_case_count()). A **line list** is aggregated first, so it needs no special handling. For a single nowcast, NULL (default) uses the tbl_now it was built from, which is only meaningful when that object still holds the later reports. A backtest instead uses the truth table it already stores.
truth_axis	Which process defines the observed counts. "report" (default) scores counts eventually reported. "revision" scores counts eventually resolved on the revision axis and requires a revision-aware truth.
truth_type	Which case type to score. Defaults to "total". Revision types such as "confirmed", "retracted", "pending", "unknown" and "net" follow the same meanings as get_latest_reported_cases() and get_latest_revised_cases(). "by_type" is refused because scoring needs one observed value per event-date/stratum target.
data	A nowcast_backtest().
...	Passed to the corresponding **scoringutils** coercion generic, most commonly forecast_unit.

Value

score_nowcast() returns a tibble with the event-date column, the strata columns, and the columns .observed, wis, ae_median, coverage_50 and coverage_90 – one row per event date and stratum.

The scoringutils::as_forecast_*() methods return the corresponding forecast_quantile, forecast_sample or forecast_point object from **scoringutils**. Each accepts a tbl_nowcast, an ensemble and a nowcast_backtest(); scoringutils::as_forecast_point() keeps the nowcast's median quantile as predicted and the resolved truth as observed. A backtest already carries the truth it was scored against, so its truth can normally be omitted.

scoringutils::as_forecast_sample() also accepts those objects when they carry posterior draws. Draws are retained by a linear_pool ensemble, but not by a quantile ensemble. A backtest retains them only when run with keep_draws = TRUE; every engine in the backtest must return draws.

References

Bracher, J., Ray, E. L., Gneiting, T., & Reich, N. G. (2021). Evaluating epidemic forecasts in an interval format. *PLoS Computational Biology*, 17(2), e1008618.

See Also

nowcast_backtest() to score many nowcasts at many now dates at once; nowcast_weights() to turn those scores into ensemble weights; get_latest_reported_cases(), which is how the truth is read off truth; nowcast_quantile_levels() for the levels being scored.

Examples

```
# A nowcast and the truth it should be judged against. Both are built by
# hand here so that the example needs no modelling package; in practice `nc`
## comes from run_nowcast() and `truth` is the same data seen later, once the
# late reports have arrived.
truth_df <- data.frame(
  onset  = rep(as.Date("2024-03-04") + 7 * (0:3), each = 3),
  report = rep(as.Date("2024-03-04") + 7 * (0:3), each = 3) + c(0, 7, 14),
  n      = c(5, 3, 2, 8, 4, 1, 6, 5, 3, 9, 2, 2)
)
truth <- tbl_now(truth_df,
  event_date = onset, report_date = report, case_count = n,
  data_type = "count-incidence", verbose = FALSE
)

# What eventually turned out to be true for each week.
get_latest_reported_cases(truth)

# A nowcast that predicted 8 / 10 / 13 for every week.
levels <- c(0.25, 0.5, 0.75)
preds <- tidyr::expand_grid(
  onset = unique(truth_df$onset), .quantile_level = levels
)
preds$.value <- rep(c(8, 10, 13), times = 4)
nc <- tbl_nowcast(predictions = preds, method = "toy", event_date = "onset")

# Lower `wis` is better. `coverage_50` says whether the truth fell inside
# the 50% interval, which it should about half the time.
score_nowcast(nc, truth = truth)

# The same comparison handed to scoringutils as a point forecast.
if (requireNamespace("scoringutils", quietly = TRUE)) {
  scoringutils::as_forecast_point(nc, truth = truth)
}

# With a real model, `truth` is the full object and the nowcast is fitted to
```

```
# a snapshot of it taken at an earlier `now`.
data(denguedat)

recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
dengue <- tbl_now(recent,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)
snapshot <- change_now(
  dplyr::filter(dengue, report_week <= as.Date("2010-10-04")),
  now = as.Date("2010-10-04")
)

if (requireNamespace("baselinenowcast", quietly = TRUE)) {
  nc <- run_nowcast(snapshot, engine_baselinenowcast(draws = 100), verbose = FALSE)
  # The FULL object is the truth: it still holds the reports that arrived
  # after the snapshot's `now`.
  score_nowcast(nc, truth = dengue)
}
```

`simulate_batch` *Inject a batch into a* `tbl_now` *by withholding and then releasing reports*

Description

[Experimental]

Simulates a reporting system that is **closed** on a given set of report dates and releases its accumulated backlog on the next open date. Reports keep their event dates and merely move *later* on the report axis, so no cases are created or destroyed – the defining property of a batch. Useful for checking that `diagnose_batches()` and `diagnose_batches2()` recover a batch you planted.

Usage

```
simulate_batch(
  x,
  closed_dates,
  held_fraction = 1,
  drop_unreleased = TRUE,
  verbose = TRUE
)
```

Arguments

`x`	A `tbl_now()` object.
`closed_dates`	A vector of report dates on which the reporting system is closed. Must be coercible to the class of the report-date column.

`held_fraction` Fraction of the reports due on each closed date that are actually **held back** (and released later); the rest report on time, so the closure is only partial. Default `1` (the whole desk is closed). With, say, `held_fraction = 0.5`, roughly half of each closed day's reports are held and half report normally. This uses the random number generator (Binomial / Bernoulli sampling), so set a seed for reproducibility. Only supported for `"linelist"` and `"count-incidence"` data (a `"count-cumulative"` total cannot be split).

`drop_unreleased` Logical; drop reports whose closed run never reopens before the end of the report axis. Default `TRUE`.

`verbose` Logical; report what was moved. Default `TRUE`.

Details

A batch is a **transport**: it moves an item's report date later while leaving its event date untouched, creating and destroying nothing. Every report whose report date lies in `closed_dates` (or, with `held_fraction < 1`, a random share of them) is re-stamped with the first report date at or after it that is *not* closed. Consequently:

- the closed dates report nothing (the **deficit**);
- the release date reports its own items *plus* the whole backlog (the **spike**);
- items released late have inflated **delays**;
- the release date draws on an unusually large number of distinct **event dates**.

All four symptoms come from the one mechanism, which is why they should not be treated as four independent pieces of evidence.

Value

A new `tbl_now` with the same event dates, strata and data type, and modified report dates.

Reports that never come back

If a closed run extends to the end of the report axis there is no open date to release into. Those reports are then unobservable – a stall that has not yet cleared is indistinguishable from data loss, an honest identification failure. `drop_unreleased = TRUE` (default) discards them, reproducing exactly what a real analyst would see.

Cumulative data

For `"count-cumulative"` data a report announces a running total. When two reports for the same event date are pushed onto the same release date, only the later one survives: it is that report date's final word on the total.

See Also

`diagnose_batches()` and `diagnose_batches2()`, the tests this exists to validate; `censor_reporting_delays_above()` for recording a real batch rather than planting one. The *Diagnosing a tbl_now* article uses this to calibrate the screen.

Examples

```
data(denguedat)

dengue_tbl <- tbl_now(
  denguedat,
  event_date  = onset_week,
  report_date = report_week,
  data_type   = "linelist",
  verbose     = FALSE
)

# Pretend the reporting desk was shut for three consecutive weeks: everything
# that would have been reported then is held, and released together afterwards.
closed <- as.Date(c("1990-06-04", "1990-06-11", "1990-06-18"))
batched_tbl <- simulate_batch(dengue_tbl, closed_dates = closed, verbose = FALSE)

# No cases are lost -- they are only moved later in the reporting process.
nrow(dengue_tbl)
nrow(batched_tbl)

# Which is the point: you now have data with a batch you planted yourself, so
# you can check whether the screen finds it.
found <- suppressWarnings(diagnose_batches(batched_tbl, lookback = 2))
found$report_date[found$batch]
```

---

surveillance_grids *The date grids* surveillance::nowcast() *needs*

Description

[Stable]

surveillance::nowcast() takes three dates and two date *grids*, and none of them have defaults you can rely on. These two helpers build the grids from the tbl_now itself, so the object stays the single source of truth for what "now" is and how wide a time step is:

- get_surveillance_when() – the dates you want **estimated**, passed as when. The most recent length steps up to and including get_now().
- get_surveillance_range() – the **whole** time axis the model is laid on, passed as control$dRange. Every step from the first event to now. Its last element is also what now itself should be: get_now() can fall mid-epoch, and surveillance::nowcast() refuses that (see below).

```
sur_fit <- surveillance::nowcast(
  now  = max(get_surveillance_range(x)),
  when = get_surveillance_when(x, length = 30),
  data = tbl_now_to_surveillance(x, verbose = FALSE),
  dEventCol = "dHospital", dReportCol = "dReport",
  control = list(dRange = get_surveillance_range(x))
)
```

Usage

```
get_surveillance_when(x, length = 30L, ..., to = NULL, by = NULL)

get_surveillance_range(x, ..., from = NULL, to = NULL, by = NULL)
```

Arguments

x	A tbl_now.
length	Number of time steps to estimate, counting back from to. The result has length elements, the last of which is to.
...	Unused, for extensibility.
to	Last date of the grid. Defaults to get_now().
by	Step, as a seq.Date() by string ("1 day", "1 week", ...). Defaults to the object's own event units.
from	First date of the grid. Defaults to the earliest event date in x.

Value

A Date vector, in increasing order.

Why dRange has to be given explicitly

Left to itself, surveillance::nowcast() infers the time axis from the data it was handed – and a **line list cannot express a zero**. A day on which nothing was reported has no rows, so it is not in the line list, so it is not in the inferred axis. That is exactly the situation at the now edge, which is the part you are nowcasting: the last few days are quiet precisely because their reports have not arrived yet, and the axis silently stops short of now. Passing dRange states the grid instead of letting it be guessed, so the quiet days at the end are modelled as zeros observed so far rather than as days that do not exist.

This is also why complete_zeroes() is no help here: it can only add zero *counts*, and a line list has no count column to put a zero in.

Which weekday the grid lands on

surveillance::nowcast() refuses a grid that does not sit at the **first day of an epoch**: a Monday for "1 week", the first of the month for "1 month". Epidemiological weeks routinely start on a Sunday instead, so both grids are snapped back to the epoch start, and both may therefore begin a few days before the dates in x. run_nowcast() shifts **surveillance**'s estimates back onto the object's own weekday when they return, so a nowcast fitted through the engine is still indexed by the event dates you gave it.

See Also

tbl_now_to_surveillance(), get_now(), get_event_units()

Examples

```
data(denguedat)
nowobj <- tbl_now(denguedat,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
get_surveillance_when(nowobj, length = 4)
range(get_surveillance_range(nowobj))
```

`tbl_now` *Create a* `tbl_now` *object*

Description

[Stable]

Surveillance data arrives late. A case that happened on Monday may only reach the surveillance system on Thursday, so counts for the most recent days always look artificially low. *Nowcasting* corrects that artifact: it estimates how many cases have already happened but have not been reported yet.

To do that, a model needs two dates for every case – when it **happened** (`event_date`) and when it was **reported** (`report_date`) – together with the date you are standing on (`now`). `tbl_now()` takes an ordinary `data.frame` and records which of its columns play those roles, so you only have to say it once.

The result still behaves like a `tibble`: `dplyr` verbs, `$`, `[` and `ggplot2` keep working, and every `tbl.now` function knows where to find the dates without being told again.

Usage

```
tbl_now(
  data,
  event_date = NULL,
  report_date = NULL,
  delay = NULL,
  strata = NULL,
  covariates = NULL,
  case_count = NULL,
  is_censored_report = NULL,
  revision_date = NULL,
  revision_type = NULL,
  revision_units = units,
  revision_levels = NULL,
  is_censored_revision = NULL,
  now = NULL,
  event_units = units,
  report_units = units,
  units = "auto",
```

```
  data_type = "auto",
  t_effects = character(0),
  verbose = TRUE,
  force = FALSE,
  warn_non_uniqueness = TRUE,
  align_weeks = FALSE,
  ...
)
```

Arguments

data	A data.frame or tibble to be converted.
event_date	tidy-select name of the column containing the event date. Optional when delay is provided together with report_date; the event date will be computed as report_date - delay.
report_date	tidy-select name of the column containing the report date. Optional when delay is provided together with event_date; the report date will be computed as event_date + delay.
delay	(optional) tidy-select or NULL (default). Name of a numeric column containing the delay (in event_units) between event_date and report_date. When provided with only one of event_date or report_date, the missing date is reconstructed from the known date and the delay. Requires units to be known (either specified via event_units or inferrable from the provided date column). Every value must be a **whole number** of those units: a fractional delay is an error, not a rounding, because a calendar has no half-days and bending the value silently moves the reconstructed date.
strata	(optional) tidy-select or NULL (default). Name of different variables (column names) in strata. Strata correspond to variables that are of interest by themselves. For example if it is of interest to generate nowcasts by gender then gender is a strata.
covariates	(optional) tidy-select or NULL (default). Name of different variables (column names) that influence the nowcast but are not strata. For example precipitation might influence a dengue nowcast but in general it is not of interest to generate nowcasts by precipitation levels.
case_count	(optional) tidy-select or NULL Name of the column with the case counts if data_type is "count-incidence" or "count-cumulative".
is_censored_report	(optional) tidy-select or NULL (default). The name of a column containing either TRUE or FALSE indicating whether the report_date is correctly specified or corresponds to a batch and thus is censored. In other words, if the report_date is accurately measured set is_censored_report = FALSE but if the report_date corresponds to an error and is only an upper bound of the real report date set is_censored_report = TRUE.
revision_date	(optional) tidy-select column holding a **third** date: the day the report was resolved. Influenza is the picture to keep in mind – symptoms begin (the event), the patient sees a doctor (the report), and days later a swab comes back. The assumed timeline is event_date <= report_date <= revision_date <= now. Leave NULL (the default) for the usual two-date object. See add_revision_date().

revision_type (optional) tidy-select column saying what the resolution *was*: "confirmed", "retracted" (it was reported, but it is not a case after all), "pending" or NA. "pending" **means reported and still waiting**, so it carries no revision date – which is a different thing from a result that was never recorded (NA). A revision date with no type warns rather than guessing, because a date alone cannot say whether the case was confirmed or retracted.

revision_units (optional) Character. Either "auto" (default), "days", "weeks", "months", "years" or "numeric" – the grid the revision date lives on, resolved the same way as report_units.

revision_levels
(optional) NULL (default) or a **named** character vector translating the labels in revision_type into the canonical outcomes, for data that was not recorded in English: c(confirmado = "confirmed", retractado = "retracted", pendiente = "pending"). The names are the labels in your data, the values are the canonical ones. The column is rewritten to the canonical values and the dictionary is kept as an attribute, readable with get_revision_levels(). Only "confirmed", "retracted", "pending" and NA are ever stored.

is_censored_revision
(optional) tidy-select or NULL (default). The revision-axis counterpart of is_censored_report: the name of a logical column marking rows whose **revision delay** is a bound rather than a measurement. Requires a revision_date. See censor_revision_delays_above().

now (optional) Date or NULL (default). The date that is considered the now of the nowcast. If no now is given then the function automatically uses the last event_date.

event_units (optional) Character. Either "auto" (default), "days", "weeks", "months", "years" or "numeric". Defaults to units.

report_units (optional) Character. Either "auto" (default), "days", "weeks", "months", "years" or "numeric". Defaults to units.

units (optional) Character. Either "auto" (default), "days", "weeks", "months", "years" or "numeric". The **default** for event_units, report_units and revision_units: say it once instead of three times. Any of the three that you give explicitly wins over units, so units = "days", report_units = "weeks" reads a daily event date against a weekly report date.

data_type (optional) Character. Either "auto", "linelist" or "count-incidence" or "count-cumulative". See section below for an explanation on data types.

t_effects (optional) Either NULL (default), a temporal_effects() object or a character vector with the names of the columns containing the temporal effects.

verbose (optional) Logical. Whether to throw a message. Default = TRUE.

force (optional) Logical. Whether to force computation overwriting pre-existing variables. Default = FALSE.

warn_non_uniqueness
(optional) Logical. Whether to throw a warning if data has several rows on the same full key: the event, report and (when declared) revision dates, the revision type, the strata, the covariates and the censoring flags. Rows carrying an NA anywhere in that key – a pending case has no revision date – are left out of the comparison, because an unobserved value cannot be shown to equal another.

`align_weeks` (optional) Logical. If both event and report units are weeks and `align_weeks = TRUE` it ensures that all weeks start in a Sunday so that week differences and `.delays` are all integer.

`...` Additional metadata to be stored as attributes on the object. Use this for provenance you want to travel with the data – `data_source`, `citation`, `population` – and read it back with `tbl_now_attributes()`.

Because anything unmatched lands here, a misspelled argument name would otherwise be accepted in silence. Names close enough to a real argument to be a typo (`case_col` for `case_count`, `stata` for `strata`) warn instead; the warning is safe to ignore if the name really was metadata.

Details

The minimum you must supply is `event_date` and `report_date` (or one of them plus a `delay` column, from which the other is reconstructed). Everything else is optional and can be added later with `add_strata()`, `add_covariates()`, `add_revision_date()` and the rest of the `add()` family.

Once the object exists the usual path is summary() to see what is in the data, `diagnose()` to see what is wrong with it, autoplot() to look at it, and `run_nowcast()` to fit a model. `vignette("tbl.now")` walks through that path end to end.

Value

An object of class `tbl_now`: the input `data` as a `tibble`, carrying extra attributes that record which columns hold the event date, report date, strata, covariates and so on, plus the `now` of the nowcast. List them with `tbl_now_attributes()`.

Attributes

The following attributes are part of a `tbl_now` and are revised by the `validate_tbl_now()` function:

event_date Name of the column refering to the event of interest.

report_date Name of the column refering to when the event of interest was reported.

strata Names of the columns corresponding to the strata (for modelling).

covariates Names of the columns corresponding to covariates (for modelling).

case_count Column containing the number of observations for that moment if `data_type` is `count-incidence` or `count-cumulative`.

temporal_effects Names of the columns refering to the temporal effects.

now Date of the `now` for a nowcast.

is_censored_report Column indicating whether the measurement is noisy (only upper bound) or not.

event_units Either `days`, `weeks`, `months`, `years` or `numeric`. Corresponds to the units of `event_date`

report_units Either `days`, `weeks`, `months`, `years` or `numeric`. Corresponds to the units of `report_date`

data_type Either `linelist`, `count-incidence` or `count-cumulative` depending on whether it is linelist data or count data with incidence (each report date's incidence) or cumulative (overall known cases at report date)

revision_date Name of the column with the (optional) third date: when the report was resolved.

revision_type Name of the column saying what that resolution was (`"confirmed"`, `"retracted"`, `"pending"`).

revision_units Units of `revision_date`, resolved like `report_units`.

revision_levels The (optional) dictionary translating the labels in `revision_type` into the canonical outcomes.

is_censored_revision Column indicating whether the *revision* delay is only a bound (the revision-axis counterpart of `is_censored_report`).

computed_temporal_effect_cols Names of the temporal-effect columns that have actually been materialised in the data by `compute_temporal_effects()`.

You can list all `tbl_now` related attributes in a specific `tbl_now` with `tbl_now_attributes()`.

Data types

The following data-types are admitted at `tbl_now` objects.

Linelist

Each row is an individual that was reported at `report_date` as happening at `event_date`.

```
df <- data.frame(
 patient     = 1:6,
 event_date  = c(rep(as.Date("2020/09/12"), 3),
                 rep(as.Date("2020/09/13"), 3)),
 report_date = c(as.Date("2020/09/12"),
                 as.Date("2020/09/13"),
                 as.Date("2020/09/14"),
                 as.Date("2020/09/13"),
                 as.Date("2020/09/14"),
                 as.Date("2020/09/15")))
print(df)
#>   patient event_date report_date
#> 1       1 2020-09-12  2020-09-12
#> 2       2 2020-09-12  2020-09-13
#> 3       3 2020-09-12  2020-09-14
#> 4       4 2020-09-13  2020-09-13
#> 5       5 2020-09-13  2020-09-14
#> 6       6 2020-09-13  2020-09-15
```

Count-incidence

Each `report_date`-`event_date` combination contains the total number of cases observed *exactly* at `report_date` for `event_date`.

```
df <- data.frame(
 n           = c(7, 1, 9, 5, 0, 2),
 event_date  = c(rep(as.Date("2020/09/12"), 3),
                 rep(as.Date("2020/09/13"), 3)),
```

```
 report_date = c(as.Date("2020/09/12"),
                 as.Date("2020/09/13"),
                 as.Date("2020/09/14"),
                 as.Date("2020/09/13"),
                 as.Date("2020/09/14"),
                 as.Date("2020/09/15")))
print(df)
#>   n event_date report_date
#> 1 7 2020-09-12  2020-09-12
#> 2 1 2020-09-12  2020-09-13
#> 3 9 2020-09-12  2020-09-14
#> 4 5 2020-09-13  2020-09-13
#> 5 0 2020-09-13  2020-09-14
#> 6 2 2020-09-13  2020-09-15
```

Count-cumulative

Each `report_date`-`event_date` combination contains the total number of cases observed up until `report_date` for `event_date`. The most recent `report_date` contains the best estimation of cases happening at `event_date`.

```
df <- data.frame(
 n           = c(1,5, 8, 2, 2, 4),
 event_date  = c(rep(as.Date("2020/09/12"), 3),
                 rep(as.Date("2020/09/13"), 3)),
 report_date = c(as.Date("2020/09/12"),
                 as.Date("2020/09/13"),
                 as.Date("2020/09/14"),
                 as.Date("2020/09/13"),
                 as.Date("2020/09/14"),
                 as.Date("2020/09/15")))
print(df)
#>   n event_date report_date
#> 1 1 2020-09-12  2020-09-12
#> 2 5 2020-09-12  2020-09-13
#> 3 8 2020-09-12  2020-09-14
#> 4 2 2020-09-13  2020-09-13
#> 5 2 2020-09-13  2020-09-14
#> 6 4 2020-09-13  2020-09-15
```

The `to_count()` function allows you to easily convert from between different data-types.

See Also

`as_tbl_now()` to convert an object created by another nowcasting package; `to_count()` to move between linelist and aggregated count data; `tbl_now_attributes()` to list what the object recorded; `validate_tbl_now()` and `diagnose()` to check it; summary() to describe it; autoplot() to plot it; `run_nowcast()` to fit a nowcast.

Examples

```
# `denguedat` is a linelist: one row per dengue case, with the week symptoms
## began (`onset_week`) and the week the case reached the surveillance system
## (`report_week`).
data(denguedat)
head(denguedat)

# Tell tbl.now which column plays which role. `now` defaults to the last
# event date seen in the data.
ndata <- tbl_now(denguedat,
  event_date = onset_week,
  report_date = report_week,
  strata = gender
)

# Printing reports back the roles it recorded, and the `now` it chose.
ndata

# A `tbl_now` is still a tibble, so ordinary manipulation works ...
ndata$newcolumn <- "something"
ndata[1:10, ]

# ... including dplyr verbs.
ndata |>
  dplyr::filter(report_week <= as.Date("1991-01-02"))

# Dropping a strata column simply forgets that stratum.
ndata |> dplyr::select(-gender)

# But dropping a column the class depends on demotes the object back to a
## plain tibble (with a warning): without an event date it can no longer
# describe a nowcast.
suppressWarnings(
  ndata |> dplyr::select(-onset_week)
)
```

---

tbl_nowcast *A nowcast produced by* run_nowcast()

Description

[Stable]

An S7 object holding a nowcast in a package-agnostic shape. It is what run_nowcast() returns and what nowcast_ensemble() and score_nowcast() consume.

Usage

```
tbl_nowcast(
```

```
  predictions = dplyr::tibble(),
  draws = NULL,
  method = NA_character_,
  fit = NULL,
  now = NULL,
  event_date = "event_date",
  strata = character(0),
  data = NULL,
  call = NULL,
  metadata = list()
)
```

Arguments

predictions	A tibble of quantile predictions. One row per (event date, stratum, quantile level) with the columns <event_date>, the strata columns, .quantile_level and .value.
draws	Either NULL (backends that only return quantiles) or a tibble with one row per (event date, stratum, draw) and the columns <event_date>, the strata columns, .draw and .value.
method	The name of the method that produced the nowcast.
fit	The untouched object returned by the backend.
now	The now of the nowcast.
event_date	Name of the event-date column in predictions/draws.
strata	Character vector with the names of the strata columns (character(0) when the nowcast is not stratified).
data	The tbl_now the nowcast was produced from.
call	The matched call of run_nowcast().
metadata	A named list with anything else the backend wants to keep.

Value

A tbl_nowcast object.

See Also

run_nowcast(), nowcast_ensemble(), score_nowcast()

Examples

```
# Normally built for you by the one-call front door, but the constructor is
# exported because a new backend -- or a test -- needs to build one directly.
predictions <- data.frame(
  onset_week = as.Date("2020-01-05") + c(0, 0, 7, 7),
  .quantile_level = c(0.5, 0.9, 0.5, 0.9),
  .value = c(10, 14, 12, 17)
)
```

```
tbl_nowcast(predictions = predictions, method = "toy", event_date = "onset_week")
```

`tbl_now_attributes` *List what a* `tbl_now` *was told about itself*

Description

[Stable]

A `tbl_now()` remembers which of its columns is the event date, which is the report date, which are strata, and so on. That information is stored as attributes on the object. `tbl_now_attributes()` shows you those, and only those – the bookkeeping attributes every `tibble` carries are left out.

Use it when you want to check what an object thinks it is, especially after a long `dplyr` pipeline.

Usage

```
tbl_now_attributes(x)
```

Arguments

x	A `tbl_now()` object.

Value

A named list of the attributes specific to the `tbl_now` class (those not shared with a plain `tibble`). Attributes are only present when they were set, so an object with no strata has no `strata` element.

See Also

`tbl_now()` and its *Attributes* section for what each attribute means; the getters for reading one attribute at a time; `change()` and `add()` for setting them; `validate_tbl_now()` to check they are coherent.

Examples

```
data(denguedat)
df_now <- tbl_now(denguedat,
  event_date = onset_week,
  report_date = report_week, strata = gender, verbose = FALSE
)

## `attributes()` returns everything, including tibble internals like `names`
# and `row.names`.
attributes(df_now) |> names()

## `tbl_now_attributes()` returns only what makes it a tbl_now.
tbl_now_attributes(df_now) |> names()
```

```
# And their values: the roles it recorded, plus the `now` of the nowcast.
tbl_now_attributes(df_now)[c("event_date", "report_date", "strata", "now")]
```

`tbl_now_baselinenowcast`

Convert between `tbl_now` *and* **baselinenowcast**

Description

[Stable]

`tbl_now_from_baselinenowcast()` accepts either the long `data.frame` (`reference_date`, `report_date`, `count`) or a `reporting_triangle` matrix (rownames = reference dates, colnames = delays, incremental counts) and converts it into a `tbl_now` of `data_type = "count-incidence"`.

`tbl_now_to_baselinenowcast()` returns either a `reporting_triangle` matrix via `baselinenowcast::as_reporting_tr` or the long `baselinenowcast`-style `data.frame`. The default is `format = "auto"`, which returns a matrix when the object has no strata and a long data frame when it does – the shape `baselinenowcast::baselinenowcast()` consumes natively in each case (with `strata_cols` naming the strata columns in the long shape). The long format also carries the **strata**, the covariates, the censoring indicator and any materialised temporal-effect columns (see `compute_temporal_effects()`); the matrix holds only the three core columns. A single reporting-triangle matrix has no strata dimension, so `format = "matrix"` on a stratified object **pools** any strata (summing the counts) with a warning.

Usage

```
tbl_now_from_baselinenowcast(
  data,
  ...,
  reference_date = "reference_date",
  report_date = "report_date",
  count = "count",
  delays_unit = NULL,
  verbose = TRUE
)

tbl_now_to_baselinenowcast(
  x,
  ...,
  format = c("auto", "matrix", "long", "triangle_list"),
  delays_unit = NULL,
  max_delay = NULL,
  complete = "auto",
  negatives = c("redistribute", "error"),
  verbose = TRUE,
  quiet = !isTRUE(verbose)
)
```

Arguments

data	A long data.frame or a reporting_triangle matrix.
...	Forwarded to as_tbl_now() (from) or baselinenowcast::as_reporting_triangle() (to, triangle formats).
reference_date, report_date, count	Column names (long format only).
delays_unit	Unit of the delay axis of the reporting triangle, one of "days" or "weeks". Both directions default to NULL, meaning it is worked out for you. For tbl_now_from_baselinenowcast() that means reading the input matrix's own delays_unit attribute (falling back to "days" when it has none); a supplied value always wins. For tbl_now_to_baselinenowcast() (triangle formats only) it is **inferred** from the object's time units when the event and report units agree and are "days" or "weeks"; otherwise you must supply it explicitly.
verbose	Logical. Print the choices that were made.
x	A tbl_now object.
format	For to, one of: • "auto" (default) – "matrix" when the object has no strata and "long" when it does. That is the shape baselinenowcast::baselinenowcast() consumes natively in each case: it takes a reporting_triangle when there is only one series to fit, and a long data.frame with a strata_cols argument when there is more than one. Pick a specific format if you need a particular return type. • "matrix" – a single baselinenowcast::as_reporting_triangle() matrix. A triangle has no strata dimension, so any strata are **pooled** (with a warning). • "long" – a tidy data frame, which can also carry the strata, covariates, temporal-effect columns and the censoring indicator. baselinenowcast::baselinenowcast() accepts this shape directly, using its strata_cols argument to name the strata columns. • "triangle_list" – one reporting triangle **per stratum**, as a tbl_now_triangle_list. Useful for inspecting each stratum's triangle; for actually fitting stratified nowcasts, the "long" shape is what **baselinenowcast** consumes natively. With no strata attached the result is still a list, of length one and named "all", so the return type never depends on whether strata happen to be present. The delay unit and the strata are taken from the object, and as_tbl_now() can rebuild a tbl_now from the result.
max_delay	Number of delay periods to keep, in the object's report units: max_delay = 30 keeps delays 0 to 29, giving a 30-column triangle. Counted exactly as tbl_now_to_epinowcast() counts it, so the same number means the same triangle in both. NULL (default) keeps every delay – which is fine on a short tail and expensive on a long one (see *Cost of a long delay tail*).
complete	For to with a triangle format: fill event periods that have no reports at all with zeroes, out to the object's get_now(), via complete_zeroes(). "auto" (the default) does this for **line-list** input only, so you do not have to remember

`to_count() |> complete_zeroes()` first. Count data is left exactly as supplied, because it *can* distinguish an observed zero from a cell that could not be observed yet (`NA`) and filling those would claim reporting was complete when it was not. `TRUE` / `FALSE` force either behaviour.

Ignored for `format = "long"`, which is a tidy data frame with no grid to complete. If you build a triangle from the long output yourself, a line list will be missing every event period in which nothing was reported – call `to_count() |> complete_zeroes()` first, or ask for `format = "triangle_list"`, which does it for you.

`negatives` How to handle the negative increments that appear when `count-cumulative` data is de-accumulated (a downward revision). `"redistribute"` (default) absorbs each negative into earlier delays with `baselinenowcast::preprocess_negative_values()`, which is what that function exists for; `"error"` refuses cumulative input instead.

`quiet` Logical. Suppress incidental output from the underlying **baselinenowcast** converter. Defaults to `TRUE` when `verbose = FALSE`.

Value

A `tbl_now` (`from`), or a `data.frame`, `reporting_triangle` or tbl_now_triangle_list (`to`), according to `format`.

Round-trip

A `reporting_triangle` distinguishes **not-yet-observed** cells (`NA`) from **observed zeros** (`0`). The `NA` cells split at the **last observed report date** (the latest report with a non-`NA` count, taken as the nowcast's `now`):

- cells with `report_date > now` are * not-yet-observable* future cells. They are **dropped** from the `tbl_now()`.
- cells with `report_date <= now` *could* have been reported but were not. They are genuinely **missing** and kept as `count = NA` rows in the `tbl_now()`.

On the way back, `baselinenowcast::as_reporting_triangle()` fills the in-triangle cells with `0` unless they are marked in the tibble as `NA`.

Sparse same-period reporting (weekly data especially)

`baselinenowcast` divides each observed row by the share of the delay distribution that should have arrived by now. When almost nothing is reported in the same period as the event, that share is tiny for the most recent row and the estimate explodes: on a weekly line list where `P(delay = 0)` is about **0.05**, a final row holding a single case became an estimate of **257 with an upper bound of 1584** against a truth of 15.

Completing the triangle to the `now` *always* leaves a final row observable only at delay 0, so no choice of cut-off avoids it. Check the delay PMF before trusting the newest rows:

```
pmf <- baselinenowcast::estimate_delay(triangle)
pmf[1]   # share expected to arrive in the same period
```

If it is small, follow `baselinenowcast`'s own advice and truncate "to an earlier reference time to ensure a nowcast, not a forecast, is being produced" – drop trailing rows whose expected observed share is below, say, 10%. Daily data with substantial same-day reporting does not have this problem.

Capping the delay axis

The triangle gets one column per delay, so a single long straggler makes it very wide and the fit very slow: capping delays at 30 days on a daily series took a fit from **314s to 50s** for a tail carrying under 1% of cases. Use `max_delay`, which counts the way `tbl_now_to_epinowcast()`'s does:

```
tbl_now_to_baselinenowcast(x, max_delay = 30)   # delays 0-29, 30 columns
```

Past a point it stops being about speed. **baselinenowcast** needs **more reference dates than delay columns** – it spends `max_delay` of them estimating the delay distribution and keeps two back for the uncertainty model – so a triangle that is as wide as it is tall cannot be fitted at all. A **snapshot ("as of") series** is exactly that shape: every snapshot restates the whole history, so the oldest event date carries a delay as long as the series and almost every cell of that width is a zero. The converter still builds the triangle; `run_nowcast()` is where it is refused, with the cap to use.

Negative delays

A reporting triangle is indexed by delay from **0**, so a report that arrived *before* its event has no cell to go in. `baselinenowcast::as_reporting_triangle()` drops it, and the cell then reads `0` – indistinguishable from an observed zero. Both triangle formats therefore **warn**, naming how many rows and cases go, so the loss is not silent; `format = "long"` is a tidy data frame with no delay axis and keeps them. `tbl_now_to_epinowcast()` drops them the same way, and warns the same way.

Filter first if you want to decide what happens:

```
x |> dplyr::filter(.delay >= 0) |> tbl_now_to_baselinenowcast()
```

Censored delays

A censoring indicator that is a property of the **case** rather than of the delay – an administrative "this date is only an upper bound" mark, say – puts a censored and an uncensored row in the same `(event_date, report_date)` cell. A reporting triangle has one slot per cell, so the extra dimension has to go before the conversion. It is removed automatically, with a warning either way:

- **count data**: the counts are summed over the flag, leaving case totals unchanged;
- **line lists**: the column is dropped, leaving one row per case.

`tbl_now_to_epidist()` is the exception and keeps the flag: estimating a delay distribution is the one job that can use it.

See Also

engine_baselinenowcast() to fit through this package rather than converting by hand; `to_count()`, because a reporting triangle needs non-negative increments and de-accumulating a revised cumulative series can produce negative ones; `complete_zeroes()` to fill the grid first. `as_tbl_now()` for the generic that dispatches to the `*_from_*()` side; `run_nowcast()`, which does the conversion for you when you fit through an `engine()`. The *One dataset, many nowcasts* article fits the same data with every supported package.

Examples

```
# Get a reporting triangle example
rt     <- baselinenowcast::example_reporting_triangle

# Convert to a tbl_now
nowobj <- tbl_now_from_baselinenowcast(rt)

## The matrix round-trip is faithful (not-yet-observed `NA` cells are kept).
identical(rt, tbl_now_to_baselinenowcast(nowobj))
```

`tbl_now_coercion_methods`

Coerce a `tbl_now` *with another package's generic*

Description

[Stable]

These S3 methods make each supported package's own coercion verb accept a `tbl_now`. They are thin wrappers around the matching `tbl_now_to_*()` converter and are quiet by default.

- `as_epidist_linelist_data()` (**epidist**) wraps `tbl_now_to_epidist()`.
- `as_epidist_aggregate_data()` (**epidist**) wraps `tbl_now_to_epidist()` with `format = "aggregate"`.
- `as_reporting_triangle()` (**baselinenowcast**) wraps `tbl_now_to_baselinenowcast()` with `format = "matrix"`.
- `as_tsibble()` (**tsibble**) wraps `tbl_now_to_tsibble()`.
- `as.data.table()` (**data.table**) wraps `tbl_now_to_data_table()`.

Usage

```
## S3 method for class 'tbl_now'
as_epidist_linelist_data(data, ..., verbose = FALSE)

## S3 method for class 'tbl_now'
as_epidist_aggregate_data(data, ..., verbose = FALSE)

## S3 method for class 'tbl_now'
as_reporting_triangle(data, ..., verbose = FALSE)

## S3 method for class 'tbl_now'
as_tsibble(x, ..., verbose = FALSE)

## S3 method for class 'tbl_now'
as.data.table(x, ..., verbose = FALSE)
```

Arguments

data, x	A tbl_now object.
...	Additional arguments forwarded to the underlying converter.
verbose	Logical; forwarded to the underlying converter. Defaults to FALSE so coercion is quiet.

Value

The object produced by the corresponding tbl_now_to_*() converter.

See Also

The tbl_now_to_*() functions these delegate to, which take the arguments: tbl_now_to_epinowcast(), tbl_now_to_baselinenowcast(), tbl_now_to_epidist(), tbl_now_to_tsibble(), tbl_now_to_data_table(); as_tbl_now() to come back the other way; as_tibble() to drop to a plain tibble.

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat[1:3000, ],
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

# These are S3 methods, so the other package's own verb works directly on a
# `tbl_now` -- no explicit converter call needed.
if (requireNamespace("tsibble", quietly = TRUE)) {
  suppressWarnings(tsibble::as_tsibble(dengue))
}

if (requireNamespace("data.table", quietly = TRUE)) {
  head(data.table::as.data.table(dengue))
}

## Use the `tbl_now_to_*()` function itself when you need its arguments; these
# methods take none beyond `verbose`.
```

---

tbl_now_data_table *Convert between* tbl_now *and* **data.table**

Description

[Stable]

tbl_now_from_data_table() converts a data.table into a tbl_now (requires explicit event_date / report_date columns). tbl_now_to_data_table() strips the tbl_now class and returns a data.table keeping every column; any lazy temporal effects are materialised first (see compute_temporal_effects()) so their columns are present.

Usage

```
tbl_now_from_data_table(data, event_date, report_date, ..., verbose = TRUE)

tbl_now_to_data_table(x, ..., verbose = TRUE)
```

Arguments

data	A data.table.
event_date, report_date	The event- and report-date columns, as tidy-select expressions: a bare column name or a string both work.
...	Forwarded to as_tbl_now() (from) or data.table::as.data.table() (to).
verbose	Logical. Print the choices that were made.
x	A tbl_now object.

Value

A tbl_now (from) or a data.table (to).

See Also

as.data.table(), the **data.table** method that calls this; as_tibble() and as_tsibble() for the other exits from the class; tbl_now() to build one from the result. as_tbl_now() for the generic that dispatches to the *_from_*() side; run_nowcast(), which does the conversion for you when you fit through an engine(). The *One dataset, many nowcasts* article fits the same data with every supported package.

Examples

```
data(denguedat)
dt <- data.table::as.data.table(denguedat)
nowobj <- tbl_now_from_data_table(dt,
  event_date = "onset_week",
  report_date = "report_week", verbose = FALSE
)
```

---

tbl_now_epidist *Convert between* tbl_now *and* **epidist**

Description

[Experimental]

epidist models the delay between a *primary* event (e.g. symptom onset) and a *secondary* event (e.g. report), storing each as an interval-censored pair of date columns: `pdate_lwr`/`pdate_upr` for the primary event and `sdate_lwr`/`sdate_upr` for the secondary event. It comes in two shapes: a one-row-per-case `epidist_linelist_data` (`epidist::as_epidist_linelist_data()`) and an `epidist_aggregate_data` that adds an `n` count column (`epidist::as_epidist_aggregate_data()`). epidist stores everything in **days** and requires every censoring window to have a strictly positive width.

`tbl_now_from_epidist()` converts either shape into a `tbl_now`:

- `"auto"` (default): use the lower bounds — `primary` (`pdate_lwr`) becomes `event_date`, `secondary` (`sdate_lwr`) becomes `report_date`. An `epidist_aggregate_data` (or any input with an `n` column) becomes `data_type = "count-incidence"` with `case_count = "n"`; otherwise `data_type = "linelist"`. The `event_units`/`report_units` are inferred from the primary censoring-window width (a 7-day window -> `"weeks"`), and a left-censored secondary window `[origin, report]` is decoded back to `is_censored_report = TRUE` with the report taken from `secondary_upper`.
- `"interval"`: instead attach the upper bounds `primary_upper` (`pdate_upr`) and `secondary_upper` (`sdate_upr`) as `covariates` (a warning is emitted).

`tbl_now_to_epidist()` performs the inverse. By default (`format = "auto"`) it builds an `epidist_aggregate_data` when `x` holds counts and an `epidist_linelist_data` otherwise, filling all four interval columns:

- the primary event spans `[event_date, event_date + w]`, where the window `w` matches the `tbl_now` unit (`"days"` -> 1 day, `"weeks"` -> 7 days, ..., or `censoring_window` if supplied);
- the secondary event spans `[report_date, report_date + w]` normally, but for rows flagged by `is_censored_report` it is left-censored to `[event_date, report_date]` (the report is known only to have happened at or before its report date, and cannot precede the event, i.e. epidist time 0) — encoding the `tbl_now` convention that a censored report is known only to have happened at or before its report date, so the window is `[event_date, report_date]`.

The strata, the covariate columns and any materialised temporal-effect columns (holidays, Fourier terms, calendar effects; see `compute_temporal_effects()`) are carried onto the epidist data unchanged, so the strata are available as covariates in an epidist model formula (epidist has no separate grouping argument).

Usage

```
tbl_now_from_epidist(
  data,
  ...,
  format = c("auto", "interval"),
  primary = "pdate_lwr",
  secondary = "sdate_lwr",
  primary_upper = "pdate_upr",
  secondary_upper = "sdate_upr",
  verbose = TRUE
```

```
)

tbl_now_to_epidist(
  x,
  ...,
  format = c("auto", "linelist", "aggregate", "interval"),
  primary_upper = NULL,
  secondary_upper = NULL,
  censoring_window = NULL,
  obs_date = NULL,
  verbose = TRUE,
  quiet = FALSE
)
```

Arguments

data	A data.frame, epidist_linelist_data or epidist_aggregate_data of **epidist** delay data.
...	Forwarded to as_tbl_now() (from) or to the relevant epidist constructor (to).
format	For from: "auto" (default) or "interval". For to: "auto" (default), "linelist", "aggregate" or "interval".
primary, secondary	Column names of the primary / secondary event lower-bound dates. Default to epidist's "pdate_lwr" / "sdate_lwr".
primary_upper, secondary_upper	Column names of the upper-bound dates. Default to epidist's "pdate_upr" / "sdate_upr". Used to infer units and decode censoring (from) or, with format = "interval", taken from covariate columns (to).
verbose	Logical. Print the choices that were made.
x	A tbl_now object.
censoring_window	(to only) Optional positive integer width, in days, of the censoring windows. If NULL (default) it is derived from the tbl_now event_units.
obs_date	(to only) Optional Date of length one (or a vector of length nrow(x)) to use as **epidist**'s obs_date column. If NULL (default) it is set to get_now(x) + censoring_window so the right-truncation clock ends at the object's own now.
quiet	Logical. A *different* channel from verbose: verbose controls the informational summary of what the conversion did, while quiet suppresses the lossy-conversion **warning** emitted by tbl_now_to_epidist().

Value

A tbl_now (from) or an epidist_linelist_data / epidist_aggregate_data object (to).

One row, one observed delay

Every row of the result is a distinct delay observation, and for the aggregate shape `n` is its weight. Two things would otherwise break that, and `tbl_now_to_epidist()` resolves both before building the object:

- **Columns the object was never told about.** covid_colombia carries `sex`, so an object built without `strata = sex` has two rows per `(notification_date, diagnosis_date)` cell. `sex` is not carried onto the epidist data, so those rows would arrive as indistinguishable duplicates; they are pooled instead, exactly as `tbl_now_to_baselinenowcast()` and `tbl_now_to_tsibble()` do. Declare the column with `add_strata()` to keep it as a model covariate rather than pool it away.
- **The revision axis**, which is dropped (see below) and therefore cannot keep two rows apart either. Pooling over it gives the `"total"` case count: every case has exactly one outcome, so no case is counted twice.

Line lists are left alone – one row is already one case – and declared strata, covariates, materialised temporal-effect columns and the `is_censored_report` flag all keep rows apart, because all of them reach the epidist object (the flag through the censoring windows).

Delays of zero, and the lognormal

A delay distribution with a **point mass at zero** cannot be fitted with a lognormal (or a gamma, or a Weibull): all have zero density at zero. If a large share of your cases are reported the same period they occur, the fit does not fail loudly – it inflates the variance until the density piles up near zero. On a daily COVID series where **57%** of cases carried a delay of exactly 0, `epidist` returned `sigma = 17.9` and an implied mean delay of `1.5e73` days.

Check before fitting:

```
mean(as.numeric(x[[get_report_date(x)]] - x[[get_event_date(x)]]) == 0)
```

If that share is large, model the delay as **discrete**, use a **zero-inflated**/hurdle form, or fit the continuous distribution to the non-zero delays and report the zero share separately.

Counts epidist cannot use

`epidist_aggregate_data` requires `n >= 1`, and so does `EpiNow2::estimate_dist()` – with the identical assertion message. Count data routinely holds rows that violate it:

- **zeros** – an `(event, report)` cell where the report added nothing, which is most cells once `complete_zeroes()` has run, and which de-accumulating a `count-cumulative` series produces wherever a cumulative total was unchanged;
- **negatives** – a cumulative total revised *downward*, which de-accumulates to a negative increment.

Both are dropped before the epidist object is built. A zero contributes no case to a delay distribution, so dropping it is lossless and is only reported when `verbose = TRUE`. A negative is not a number of cases at all, so dropping it discards the revision and **warns**. If nothing usable is left the conversion aborts saying so, rather than letting **epidist**'s own `Assertion on 'data$n' failed` through.

Model choice for count data

`epidist::as_epidist_marginal_model()` is the model built for aggregated counts: it works from the `(delay, observation time)` cells the converter produces, so a month of cases costs a few hundred *weights* rather than a few thousand rows. The latent and naive models are alternatives that expand the counts back to one row per case.

The `now` and the observation window

epidist uses `obs_date` (an "observation stopped at" instant) to correct for right truncation: any case with an event date near the end of the series is under-observed, because there was less time for its report to arrive. `tbl_now_to_epidist()` sets `obs_date <- get_now(x) + w` (the end of the `now` period, widened by the censoring window `w`) so the truncation clock ends at the object's own `now` rather than at the last reported case. The two are usually the same on a fully-observed series and can differ when the tail is silent or when `change_now()` moves `now` forward for a backtest. Pass `obs_date` explicitly to override.

`tbl_now_from_epidist()` reads the same column back on the `"auto"` path: `now` on the returned `tbl_now` is `max(obs_date) - w`, so a round trip preserves it (up to the censoring-window widening).

Revision axis (not modelled)

epidist estimates one delay distribution – the primary-to-secondary delay, which the converter maps to `event_date -> report_date`. It has no way to represent the revision axis, so `has_revision(x)`, `revision_type`, `is_censored_revision` and the revision dates are **dropped** from the epidist object. `tbl_now_to_epidist()` warns once when it drops them, so a user who declared a revision process is told the converter is not surfacing it. Count rows that differed only in their revision date or outcome are then pooled, so the drop does not leave duplicate rows behind.

See Also

add and revision_delay, since **epidist** is about delay distributions and a `tbl_now` may carry two of them; `censor_reporting_delays_above()` for the long delays that would otherwise dominate a fitted distribution; tidy() for the fitted result. `as_tbl_now()` for the generic that dispatches to the `*_from_*()` side; `run_nowcast()`, which does the conversion for you when you fit through an `engine()`. The *One dataset, many nowcasts* article fits the same data with every supported package.

Examples

```
## --- Linelist epidist data (one row per case) ---
ll <- suppressMessages(epidist::as_epidist_linelist_data(
  data.frame(
    pdate_lwr = as.Date(c("2020-03-01", "2020-03-02", "2020-03-02")),
    sdate_lwr = as.Date(c("2020-03-05", "2020-03-04", "2020-03-06"))
  ),
  pdate_lwr = "pdate_lwr", sdate_lwr = "sdate_lwr"
))
# -> a linelist tbl_now ...
nowll <- tbl_now_from_epidist(ll, verbose = FALSE)
get_data_type(nowll)
# ... and back to an epidist_linelist_data
tbl_now_to_epidist(nowll, verbose = FALSE, quiet = TRUE)
```

```
## --- Aggregate epidist data (counts in an `n` column) ---
agg <- suppressMessages(epidist::as_epidist_aggregate_data(
  data.frame(
    pdate_lwr = as.Date(c("2020-03-01", "2020-03-02")),
    sdate_lwr = as.Date(c("2020-03-05", "2020-03-04")),
    n = c(7, 3)
  ),
  n = "n", pdate_lwr = "pdate_lwr", sdate_lwr = "sdate_lwr"
))
## -> a count-incidence tbl_now (case_count = "n") ...
nowagg <- tbl_now_from_epidist(agg, verbose = FALSE)
get_data_type(nowagg)
## ... and back to an epidist_aggregate_data (auto-detected from the counts)
tbl_now_to_epidist(nowagg, verbose = FALSE, quiet = TRUE)
```

`tbl_now_EpiNow2` *Convert between* `tbl_now` *and* **EpiNow2**

Description

[Experimental]

EpiNow2 takes several different input shapes, one per entry point, so `tbl_now_to_EpiNow2()` is told which one you want with `target` – named after the **EpiNow2** function the result is passed to, so it can be handed over unchanged:

`"estimate_infections"` a `data.frame` of `date` / `confirm`, the series as known at `get_now()`. Also what `EpiNow2::epinow()` takes.

`"regional_epinow"` the same, plus a `region` column built from the object's strata.

`"estimate_truncation"` a tbl_now_epinow2_snapshots list – one `date`/`confirm` snapshot per report date, which is the one **EpiNow2** model that uses the report dimension a `tbl_now` exists to carry.

`"estimate_secondary"` a `data.frame` of `date` / `primary` / `secondary`, where `primary` counts reported arrivals by `report_date` and `secondary` counts resolved revisions by `revision_date`, filtered by `secondary_type`. This is a **repurposing** of `EpiNow2::estimate_secondary()`: the model was written for two epidemiological streams linked by a delay (cases and deaths, say), and here the two streams are one series and its own revisions, so the fitted delay is report-to-revision. The converter warns about the repurposing when it runs.

`"estimate_dist"` the interval-censored `pdate_lwr` / `pdate_upr` / `sdate_lwr` / `sdate_upr` / `obs_date` frame that `EpiNow2::estimate_dist()` fits a **delay distribution** to (new in **EpiNow2** 1.9.0). Count data rides along as the `n` weight column. `estimate_dist()` vendors likelihood functions from primarycensored, and its help asks that you cite **primarycensored** alongside **EpiNow2** when using it (`citation("primarycensored")`). Like `tbl_now_to_epidist()`, this target returns one row per observed delay: the revision axis is dropped and undeclared columns are pooled, because neither reaches `estimate_dist()` and so neither can keep two rows apart.

`tbl_now_from_EpiNow2()` inverts the snapshot form: snapshot k is the series as known at report date k, so differencing consecutive snapshots recovers `count-incidence` exactly. There is deliberately **no** inverse for the other targets: a single series has no report dimension to recover, a secondary stream is already aggregated, and a delay distribution is not case data.

Usage

```
tbl_now_to_EpiNow2(
  x,
  ...,
  target = c("estimate_infections", "regional_epinow", "estimate_truncation",
    "estimate_secondary", "estimate_dist"),
  snapshots = NULL,
  secondary_type = c("confirmed", "total", "retracted", "unknown"),
  accumulate = "auto",
  complete = "auto",
  verbose = TRUE,
  quiet = FALSE
)

tbl_now_from_EpiNow2(data, ..., report_dates = NULL, verbose = TRUE)
```

Arguments

`x`	A `tbl_now` object.
`...`	Forwarded to `as_tbl_now()` (`from`); unused (`to`).
`target`	Which **EpiNow2** entry point the result is for. See above.
`snapshots`	For `"estimate_truncation"`: how many snapshots to emit, taken from the **latest** report dates. `NULL` (default) uses 5, matching `EpiNow2::example_truncated`. One snapshot per distinct report date is usually far more than the model can fit.
`secondary_type`	For `"estimate_secondary"`: which revision outcomes to count in the `secondary` stream. One of `"confirmed"` (default), `"total"`, `"retracted"` or `"unknown"`. Pending cases are not on the revision axis, and `"net"` can be negative, which `EpiNow2::estimate_secondary()` cannot represent as a count stream.
`accumulate`	How to handle non-daily data. `"auto"` (default) lays a weekly series on **EpiNow2**'s daily grid with an `accumulate` column; `FALSE` passes the rows through unchanged, which is almost always wrong (see *Non-daily data*). Ignored for `"estimate_dist"`, which works in censoring windows rather than on a grid.
`complete`	For the series targets: fill event periods that have no reports at all with zeroes, out to the object's `get_now()`, via `complete_zeroes()`. `"auto"` (the default) does this for **line-list** input only. A line list has no row for a period in which nothing was reported, so a series built from one stops at the last period that *has* a report – short of the `now`, which is the period the nowcast is about. Count data is left exactly as supplied, because it can say "observed zero" itself. `TRUE` / `FALSE` force either behaviour; `TRUE` on `count-cumulative` input de-accumulates it first. Ignored for `"estimate_dist"`, which works in censoring windows rather than on a grid.

verbose	Logical. Print the choices that were made.
quiet	Logical. A *different* channel from verbose: verbose controls the informational summary of what the conversion did, while quiet suppresses the lossy-conversion warning. Set both to keep a conversion entirely silent.
data	A tbl_now_epinow2_snapshots, or a plain list of date/confirm data frames (e.g. EpiNow2::example_truncated), in which case report_dates is required.
report_dates	For from: a Date vector, one per snapshot, saying when each was taken. Read from the object's attribute when it has one.

Value

For to, a data.frame or a tbl_now_epinow2_snapshots, according to target. For from, a tbl_now of data_type = "count-incidence".

Non-daily data

EpiNow2 models a **daily** process. As of 1.9.0 there is no timestep, interval or period argument on any of its entry points, so a weekly series passed as one row per week is read as one row per **day** and the fit is silently wrong on the time axis – no error, just an epidemic seven times too fast.

Its own answer is the accumulate column (see EpiNow2::fill_missing()): the series is laid on a daily grid and the filler days are marked to be added to the next real observation. accumulate = "auto" does this from get_event_units() for case-count targets, and from the shared report_units / revision_units grid for estimate_secondary. Units coarser than a week, and the "numeric" grid, are refused outright rather than approximated.

What EpiNow2 will not take

- EpiNow2::estimate_delay() takes a bare vector of delays. Its own help now points at estimate_dist() as "the recommended replacement", and it throws away the censoring a tbl_now carries, so there is no target for it either. If you want it anyway, it is x$.delay.

See Also

tbl_now_to_epidist(), which builds the same censoring windows as target = "estimate_dist" – the two are different front ends onto one delay-distribution schema.

Examples

```
data(denguedat)
nowobj <- tbl_now(denguedat[1:2000, ],
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
## A single daily series for estimate_infections() -- the weekly data is laid
# on EpiNow2's daily grid.
head(tbl_now_to_EpiNow2(nowobj, verbose = FALSE, quiet = TRUE))

## Snapshots for estimate_truncation(), which uses the report dimension.
snaps <- tbl_now_to_EpiNow2(nowobj,
  target = "estimate_truncation", verbose = FALSE, quiet = TRUE
```

```
)
snaps
```

`tbl_now_epinow2_snapshots`

Snapshots of one series, as **EpiNow2** *estimates truncation from*

Description

[Stable]

The object returned by `tbl_now_to_EpiNow2(x, target = "estimate_truncation")`: a list of `date`/`confirm` data frames, one per report date, plus the report dates themselves so the object can be turned back into a `tbl_now`.

It is a **thin** class – still a list, so it can be handed to `EpiNow2::estimate_truncation()` unchanged:

```
snaps <- tbl_now_to_EpiNow2(x, target = "estimate_truncation")
EpiNow2::estimate_truncation(snaps)
```

The class exists because a bare list of `date`/`confirm` frames does not say *when* each snapshot was taken, and without that the reporting triangle cannot be recovered from it. Printing also distinguishes it from the superficially similar list `EpiNow2::estimate_secondary()` does *not* take.

Usage

```
## S3 method for class 'tbl_now_epinow2_snapshots'
print(x, ...)
```

Arguments

`x`	A `tbl_now_epinow2_snapshots`.
`...`	Ignored.

Value

`print()` returns `x` invisibly.

See Also

`tbl_now_to_EpiNow2()`, `as_tbl_now()`

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat[1:3000, ],
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

# A stack of snapshots: what the series looked like at each of several past
## report dates. EpiNow2::estimate_truncation() uses these to learn how much
# the most recent counts are still going to grow.
snaps <- tbl_now_to_EpiNow2(dengue,
  target = "estimate_truncation", verbose = FALSE, quiet = TRUE
)

# Printing summarises the stack rather than dumping every snapshot.
snaps

length(snaps)
head(snaps[[1]])
```

---

tbl_now_epinowcast *Convert between* tbl_now *and* **epinowcast**

Description

[Stable]

epinowcast represents the same observations in several shapes:

- the **raw long input** `data.frame` (`reference_date`, `report_date` and a **cumulative** `confirm` column, plus optional grouping columns) consumed by `epinowcast::enw_preprocess_data()`;
- the **preprocessed object** returned by `epinowcast::enw_preprocess_data()` (a nested `data.table` used downstream for modelling, summaries and plotting);
- a fitted `epinowcast` object (which extends the preprocessed object).

`tbl_now_from_epinowcast()` accepts **any** of these and converts the cumulative observations into a `tbl_now` of `data_type = "count-cumulative"`. When given a preprocessed or fitted object, the grouping (by) columns are detected automatically and the observations are those retained by preprocessing (i.e. truncated at `max_delay`).

`tbl_now_to_epinowcast()` takes a `tbl_now` and, by default, builds the preprocessed `epinowcast::enw_preprocess_data` object (the representation used for epinowcast's summaries and plots). With `preprocess = FALSE` it returns the completed long observation `data.table` (the model *input* format, as produced by `epinowcast::enw_complete_dates()`).

Usage

```
tbl_now_from_epinowcast(
  data,
  ...,
  reference_date = "reference_date",
  report_date = "report_date",
  confirm = "confirm",
  strata = NULL,
  verbose = TRUE
)

tbl_now_to_epinowcast(
  x,
  ...,
  max_delay = NULL,
  timestep = NULL,
  missing_reference = NULL,
  preprocess = TRUE,
  verbose = TRUE,
  quiet = FALSE
)
```

Arguments

data	Source data: a raw long data.frame/data.table, an enw_preprocess_data object, or a fitted epinowcast object.
...	Additional arguments forwarded to as_tbl_now() (for from) or to epinowcast::enw_preprocess_data (for to).
reference_date, report_date, confirm	Column names (raw input only; ignored for preprocessed/fitted objects).
strata	Optional character vector of grouping columns. If NULL (default) the grouping is taken from the preprocessed object's by, or, for raw input, any column other than reference_date, report_date and confirm.
verbose	Logical. Print the choices that were made.
x	A tbl_now object.
max_delay	Maximum delay (in timesteps) to use when preprocessing. If NULL it is inferred from the data as max(.delay) + 1. Because .delay is measured in the object's report units, this is only in timesteps when timestep matches those units — which is what the default infers. The cap is applied by epinowcast::enw_preprocess_data() after the full observed delay history has been completed, so observations beyond the cap still contribute to epinowcast's max_confirm calculation.
timestep	The **epinowcast** timestep: "day", "week", or a whole number of days. NULL (default) infers it from the object's report units ("days" -> "day", "weeks" -> "week"), which keeps max_delay and the temporal-effect covariates on the same grid as the data. Other units cannot be inferred (**epinowcast** does not support calendar months) — pass a number of days explicitly, e.g. timestep = 28.

missing_reference
: Passed to epinowcast::enw_complete_dates(). NULL (default) uses TRUE when the tbl_now contains reports with a missing event/reference date and FALSE otherwise. This preserves real missing-reference reports without synthesising padding rows when none were observed. Supply TRUE or FALSE to override the detection.

preprocess
: If TRUE (default) returns an enw_preprocess_data object; if FALSE returns the completed observation data.table.

quiet
: Logical. A *different* channel from verbose: verbose controls the informational summary of what the conversion did, while quiet suppresses the lossy-conversion **warning** emitted by tbl_now_to_epinowcast() (see the Round-trip section).

Value

tbl_now_from_epinowcast() returns a tbl_now. tbl_now_to_epinowcast() returns an enw_preprocess_data object or a data.table.

Round-trip

The round-trip is **not** the identity, and tbl_now_to_epinowcast() warns to that effect (silence it with quiet = TRUE). If you already have the data in epinowcast's format, work from it directly rather than converting through tbl_now and back.

tbl_now_from_epinowcast(tbl_now_to_epinowcast(x)) recovers x up to the max_delay truncation that epinowcast applies during preprocessing: reports with a delay beyond max_delay are dropped by epinowcast::enw_preprocess_data() and so are absent from the result.

Conversely, tbl_now_to_epinowcast(tbl_now_from_epinowcast(pobs)) is not identical to pobs, because a tbl_now does not retain everything an enw_preprocess_data object carries:

- **Covariate columns** that are neither the core reference_date/report_date/confirm, a grouping (by) column, nor a materialised temporal-effect column are dropped. The temporal-effect columns (holidays, Fourier terms, calendar effects) *are* carried over: the lazy temporal_effects() spec is materialised with compute_temporal_effects() and the resulting columns are passed through to the observations and metareference/metareport tables.
- **Grouping indices** (.group) are reassigned from the factor levels, so the row order of the nested tables can differ even though the underlying values match.
- **NA-reference padding** is not regenerated by default (see missing_reference).
- max_confirm (and the derived cum_prop_reported) will not match for reference dates whose reporting completes *after* max_delay. The modelled confirm (the reporting triangle) is truncated at max_delay, but epinowcast computes max_confirm as the eventual final total from the *untruncated* history. A tbl_now only stores the truncated triangle, so reports arriving beyond max_delay are gone: on the way back epinowcast::enw_preprocess_data() recomputes max_confirm from the within-window data and obtains a smaller value. The confirm counts themselves still round-trip exactly; only these truncation-derived summary columns differ. (For example, in germany_covid19_hosp the 2021-04-06 / 00-04 cell reaches 7 by delay 40 but a final 11 only at delay 74, so its max_confirm is 11 in pobs and 7 after the round-trip.)

Per-cell observation flags

A `tbl_now` `is_censored_report` flag records an upper bound on the report date of an individual case, and it is per-case: it can differ between two rows in the same `(event_date, report_date)` cell. **epinowcast**'s preprocessed object has no equivalent – it stores one cumulative count per cell – so the converter collapses the flag before conversion (summing the counts over it for count data, dropping the column for a line list), with a warning.

epinowcast's nearest concept is not equivalent, but is worth knowing. `epinowcast::enw_obs()` takes an `observation_indicator` naming a *per-cell* logical column that marks cells as observed or not, which epinowcast then uses to decide whether a cell contributes to the likelihood. It is a **cell** flag rather than a **case** flag, so it cannot be built from `is_censored_report` alone: two rows in the same cell can disagree, and a cell-level column has to pick one answer. If you have a genuinely cell-level "known unobservable" signal you can add a column to the preprocessed object by hand and reference it with `obs = enw_obs(observation_indicator = "...")` in the fit.

Negative delays

A reporting triangle is indexed by delay from **0**, so a report that arrived *before* its event has no cell to go in. `baselinenowcast::as_reporting_triangle()` drops it, and the cell then reads `0` – indistinguishable from an observed zero. Both triangle formats therefore **warn**, naming how many rows and cases go, so the loss is not silent; `format = "long"` is a tidy data frame with no delay axis and keeps them. `tbl_now_to_epinowcast()` drops them the same way, and warns the same way.

Filter first if you want to decide what happens:

```
x |> dplyr::filter(.delay >= 0) |> tbl_now_to_baselinenowcast()
```

Censored delays

A censoring indicator that is a property of the **case** rather than of the delay – an administrative "this date is only an upper bound" mark, say – puts a censored and an uncensored row in the same `(event_date, report_date)` cell. A reporting triangle has one slot per cell, so the extra dimension has to go before the conversion. It is removed automatically, with a warning either way:

- **count data**: the counts are summed over the flag, leaving case totals unchanged;
- **line lists**: the column is dropped, leaving one row per case.

`tbl_now_to_epidist()` is the exception and keeps the flag: estimating a delay distribution is the one job that can use it.

See Also

engine_epinowcast() to fit through this package rather than converting by hand; `align_weeks()`, because **epinowcast** lays its grid out in whole timesteps; `complete_zeroes()` to fill the grid; tidy() for the fitted result. `as_tbl_now()` for the generic that dispatches to the `*_from_*()` side; `run_nowcast()`, which does the conversion for you when you fit through an `engine()`. The *One dataset, many nowcasts* article fits the same data with every supported package.

Examples

```
library(data.table)
library(epinowcast)

## CRAN asks examples to use at most two cores; data.table would otherwise
## take every one it can find.
data.table::setDTthreads(2)

# epinowcast's own example data: German COVID-19 hospitalisations by age.
obs <- germany_covid19_hosp[location == "DE"][, location := NULL]

## A few weeks and a short delay keep the example quick; preprocessing the whole
## series with `max_delay = 40` costs about ten times as much CPU.
recent <- obs[reference_date >= as.Date("2021-10-15")]
pobs <- epinowcast::enw_preprocess_data(recent, max_delay = 10, by = "age_group")

# From the data.table input format ...
nowobj <- tbl_now_from_epinowcast(recent, strata = c("age_group"))
nowobj

# ... or from a preprocessed epinowcast object.
tbl_epi <- tbl_now_from_epinowcast(pobs)

# And back out again.
preprocessed_tbl <- tbl_now_to_epinowcast(tbl_epi, quiet = TRUE)
```

---

tbl_now_nobbs *Convert a* tbl_now *into the line list* **NobBS** *nowcasts from*

Description

[Stable]

NobBS::NobBS() counts **rows**: it takes an individual-level line list with one column for the event date and one for the report date, and treats each row as a case. Handing it `count-incidence` data directly is therefore silently wrong – a table of 1,174 rows carrying 50,160 cases is nowcast as 1,174 cases. This converter expands counts to one row per case first, so the totals NobBS sees are the totals in your data.

Trim **before** converting when the series is long: the expansion is one row per case, and NobBS()'s own `moving_window` only limits what it *fits*, not what it is handed.

Usage

```
tbl_now_to_nobbs(
  x,
  ...,
  event_col = "onset_date",
  report_col = "report_date",
```

```
  strata_col = "strata",
  strata_sep = " | ",
  verbose = TRUE
)
```

Arguments

x	A tbl_now.
...	Unused, for extensibility.
event_col, report_col	Names the two date columns should take in the result. The defaults match the arguments of NobBS::NobBS().
strata_col	Name of the single stratifying column to add, holding every declared stratum pasted together. This is what NobBS::NobBS.strat()'s own strata argument takes. NULL leaves it out. Ignored when the object declares no strata.
strata_sep	Separator used to paste the strata into strata_col.
verbose	Print what the conversion did. The units line prints the string NobBS::NobBS() itself accepts ("1 day" or "1 week"), not the object's own "days" / "weeks", so it can be pasted straight into the call.

Value

A data.frame with one row per case, ready for NobBS::NobBS(). The strata, covariates and temporal-effect columns ride along, plus the single pasted strata_col that NobBS::NobBS.strat() takes.

Stratified nowcasts

NobBS::NobBS.strat() fits one nowcast per stratum, and its strata argument names **one** column. A tbl_now may declare several – age group and region, say – and "nowcast each age-group-and-region separately" is a single stratum as far as NobBS is concerned. So the declared columns are also pasted into one strata column, which you hand straight to NobBS.strat():

```
nb <- tbl_now_to_nobbs(x, verbose = FALSE)
NobBS::NobBS.strat(nb, now = get_now(x), units = "1 day",
                   onset_date = "onset_date", report_date = "report_date",
                   strata = "strata")
```

The original columns are kept alongside it, so a hand-rolled per-stratum loop can still split on them. Choose a separator your stratum values do not contain: run_nowcast() splits the label back into the original columns when it tidies the fit.

Units NobBS can model

NobBS::NobBS() documents units as "1 day" or "1 week" and nothing else, so this converter aborts on any other grid rather than hand back a line list **NobBS** cannot use. That includes a "numeric" grid: its date columns are integer indices, and coercing them with as.Date() would anchor them at the 1970 epoch and return a plausible-looking line list of invented dates. Aggregate to days or weeks first (see align_weeks()).

Censored delays

A censoring indicator that is a property of the **case** rather than of the delay – an administrative "this date is only an upper bound" mark, say – puts a censored and an uncensored row in the same `(event_date, report_date)` cell. A reporting triangle has one slot per cell, so the extra dimension has to go before the conversion. It is removed automatically, with a warning either way:

- **count data**: the counts are summed over the flag, leaving case totals unchanged;
- **line lists**: the column is dropped, leaving one row per case.

`tbl_now_to_epidist()` is the exception and keeps the flag: estimating a delay distribution is the one job that can use it.

See Also

`tbl_now_to_surveillance()`, `tbl_now_to_epinowcast()`

Examples

```
data(denguedat)
nowobj <- tbl_now(denguedat,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
nb <- tbl_now_to_nobbs(nowobj, verbose = FALSE)
head(nb)
```

---

tbl_now_palette *The* `tbl.now` *colour palette*

Description

Builds the named colour palette every `plot_*()` function, `autoplot()` and `diagnostic_plot()` draw from. Each element is named for the **role** it plays in a plot, never for its hue, so a palette in different colours is a matter of overriding the roles you care about:

```
plot_reporting_triangle(x, palette = tbl_now_palette(reporting = "#5B4B8A"))
```

Arguments you do not name keep the package default, so a partial palette is always complete. Passing a bare named vector works too, as long as it carries every role below – the plots validate it and name what is missing.

Usage

```
tbl_now_palette(
  reporting = "#B85348",
  reporting_light = "#e78b7f",
  epidemic = "#5F7E62",
  epidemic_light = "#A8BFA9",
  epidemic_mid = "#7A9E7E",
  epidemic_dark = "#334335",
  revision = "#C79800",
  revision_light = "#E6CE80",
  retracted = "#3E6F9E",
  ink = "#262626",
  ink_muted = "#607060",
  ink_inverse = "#FFFFFF",
  surface = "#FFFFFF",
  surface_muted = "#F5F5F5",
  surface_dark = "#1A1A1A",
  grid_major = "#999999",
  grid_minor = "#E0E0E0",
  guide = "#D9D9D9",
  guide_strong = "#737373",
  annotation = "#333333",
  neutral = "#B3B3B3",
  zero = "#C4D5DE",
  pending = "#C9CEC9",
  observed = "#DFE1DF"
)
```

Arguments

`reporting`	Strong colour of the reporting process (bars, medians, flagged points).
`reporting_light`	Attenuated reporting colour (box fills, wide intervals).
`epidemic`	Strong colour of the epidemic process (lines, bars).
`epidemic_light`	Attenuated epidemic colour (area fills).
`epidemic_mid`	Mid-tone epidemic colour (the middle stop of the count ramp).
`epidemic_dark`	Darkest epidemic colour (dense overplotted curves).
`revision`	Strong colour of the revision process, and of a `confirmed` outcome on it.
`revision_light`	Attenuated revision colour (box fills).
`retracted`	A report the revision process took back. The counterpart of `revision`, so that the two resolutions can be told apart at a glance.
`ink`	Body text, axis text and titles.
`ink_muted`	Secondary text: subtitles, captions, immature-region shading.
`ink_inverse`	Text drawn *on top of* a filled label.
`surface`	Fill of a label or a highlight drawn over the data.

surface_muted	Palest surface: the low end of a sequential ramp.
surface_dark	Deep surface for a region with no estimate.
grid_major	Major gridlines the package draws itself.
grid_minor	Minor gridlines the package draws itself.
guide	Weak reference lines (a zero line, the low end of a count ramp).
guide_strong	Stronger reference lines (the reporting triangle's iso-report diagonals).
annotation	Text of an annotation label drawn over the data.
neutral	De-emphasised marks: the points a test did *not* flag.
zero	A cell that is observable but reported nothing – a genuine zero, as opposed to a cell that is blank because it is not yet reportable.
pending	A case that is reported and not yet resolved.
observed	Counts as they stand now, drawn underneath an estimate of them. Deliberately neutral: colouring these with a process role would put the data in the same visual family as the model fitted to it.

Value

A named character vector of colours, one per role, with class `tbl_now_palette`.

The grammar

The package has one visual grammar and the role names state it:

- `reporting*` – the *reporting* process: report dates, delays, anything about **when we found out**. Red by default.
- `epidemic*` – the *epidemic* process: event dates, case counts, anything about **what happened**. Green by default.
- `revision*` – the *revision* process: revision dates and resolution arrivals. Ochre by default.

A palette that swaps the two hues is fine; a *plot* that draws delays with an `epidemic*` role is a bug, whatever colour it comes out.

The remaining roles are furniture — text, gridlines, reference lines and the three data states that are not a process (`zero`, `pending`, `observed`).

Revision outcomes

A panel that splits the revision axis by outcome (see the `by_revision_type` argument of `plot_delay_distribution()` and `plot_reporting_process()`) draws `confirmed` in the `revision` colour, `pending` in `surface`, `retracted` in `retracted` and an unrecorded outcome in `neutral`. Ochre and blue are the two resolutions, white is the case that has not resolved, grey the one whose outcome was never written down.

See Also

autoplot(), `diagnostic_plot()` and `plot_reporting_hexamap()`, all of which take a `palette` argument.

Examples

```
tbl_now_palette()

# Override one role; the rest keep the package defaults.
tbl_now_palette(reporting = "#5B4B8A")

data(denguedat)
dn <- tbl_now(denguedat, onset_week, report_week, verbose = FALSE)
plot_epidemic_process(dn, palette = tbl_now_palette(epidemic = "#2F6DB4"))
```

---

tbl_now_summary *Summarise a* tbl_now

Description

[Experimental]

`summary()` describes a `tbl_now` the way a nowcaster needs it described: how many cases arrive on each of the object's time axes, how long they take to get there, how sparse the series is, what fraction of the data is censored or still pending, and how far the object reaches.

Every block of the summary is also available on its own – see nowcast_summary_components – and `summary()` is exactly the `dplyr::bind_rows()` of those pieces.

Usage

```
## S3 method for class 'tbl_now'
summary(object, ..., by_strata = NULL, strata = NULL, growth_k = 7)
```

Arguments

`object`	A `tbl_now` object.
`...`	Unused, for compatibility with the `base::summary()` generic.
`by_strata`	Logical. Add one set of rows per stratum on top of the pooled (`"all"`) rows. Defaults to `TRUE` when the object has strata.
`strata`	Character vector of columns to stratify by. Defaults to `get_strata(object)`.
`growth_k`	Number of delays for the cumulative growth rows.

Details

The date grids. "Cases per event date" is a statement about a *calendar*, not about the rows present in the data, so each axis is completed to a full grid running from the earliest observed date on that axis to `get_now()`, stepping by that axis's units. Dates with no rows count as zeros. This is what makes `prop_zero` and the zero-run lengths meaningful, and it is why a line list – which cannot represent a zero – is summarised correctly here.

Not-yet-observed cells are dropped. An `NA` count means the cell has not been observed yet, unlike a `0`, which was observed and was zero. Such rows carry no cases, so they are excluded rather

than allowed to turn every total they touch into NA. How many were dropped is reported as the "unobserved_cells" coverage row.

The grid is **global**: when by_strata = TRUE every stratum is summarised on the same grid, so a stratum whose cases start late genuinely shows the leading zeros. Otherwise the strata would not be comparable.

Count-cumulative data gets no delay rows. A cumulative total is not additive across delays, so a case-weighted delay distribution would be meaningless. The "growth" rows take their place, describing how each event date's total grows from one delay to the next. Call to_count(x, to = "count-incidence") first if you want the delay distribution – and note that de-accumulating can produce negative increments.

Value

A tibble with the columns described above.

The columns

Every function in this family returns the same schema, so results can be stacked with dplyr::bind_rows() and filtered with dplyr::filter().

component Which block the row belongs to: "cases", "delay", "zero_run", "composition", "growth" or "coverage".

quantity What the row describes, including the category for the compositional rows ("revision_type = confirmed").

stratum Which subset of the data the row describes: "all" for the pooled rows, or the stratum label otherwise.

n How many **things of the block's own kind** the row counts. It is never a case count, and what it counts changes with the block: dates on the grid for "cases", runs of zeros for "zero_run", and (event date, report date) cells for "delay" and "composition".

total How many **cases** are behind the row: records for a line list, the sum of the case-count column otherwise. So n and total answer different questions and are equal only when every cell holds exactly one case. In the "composition" block, for instance, n is how many cells carry that category and total is how many cases do – and prop is computed from total, the cases.

mean, sd Mean and standard deviation. For the case-weighted rows these are the weighted versions, equal to what you would get by expanding the counts to one row per case.

min, q25, q50, q75, q90, max Quantiles. See the note below on which estimator is used.

prop_zero Proportion of dates on the grid that are exactly zero.

prop Proportion of cases in this category (compositional rows).

value A single scalar that is not a distribution: a gap or an occupancy. The "growth" rows are distributions over event dates, so they populate mean/sd/the quantiles instead and leave value empty.

date_min, date_max Date range. Present only when the result contains "coverage" rows.

unobserved_cells A "coverage" row counting the NA-count rows excluded as not yet observed.

Note

Quantiles are inverse-ECDF (type 1), not `stats::quantile()`'s default. `q50` is the smallest value whose cumulative weight reaches `0.5`, which for an even number of observations is the upper of the two middle values rather than their average. This is deliberate: it is the same estimator `autoplot.tbl_now()` and `diagnose_drift()` use for the delay quantiles they draw, so the numbers in this table match the numbers in the plots. It also always returns a value that was actually observed, which a half-case delay is not.

See Also

nowcast_summary_components for the individual blocks.

Examples

```
data(denguedat)
# The last five years. The full twenty-year series gives the same shape
# of answer, it just takes longer to compute.
recent <- denguedat[denguedat$onset_week >= as.Date("2006-01-01"), ]
ndata <- tbl_now(recent,
  event_date = "onset_week",
  report_date = "report_week",
  strata = "gender",
  verbose = FALSE
)

# The whole summary: one row per quantity, per stratum.
overview <- summary(ndata)
overview

# It is an ordinary tibble, so pick out the block you want.
overview |> dplyr::filter(component == "delay")

# `n` and `total` are different questions. In the compositional block `n`
# counts the event-report cells carrying the category and `total` counts
# the cases in them.
overview |>
  dplyr::filter(component == "composition") |>
  dplyr::select(quantity, stratum, n, total, prop)

# Pooled rows only, ignoring the strata.
summary(ndata, by_strata = FALSE)
```

`tbl_now_surveillance` *Convert a* `tbl_now` *into the line list* **surveillance** *nowcasts from*

Description

[Stable]

surveillance::nowcast() works from an individual-level line list with one column holding the event date and another the report date, named by its dEventCol / dReportCol arguments. tbl_now_to_surveillance() produces exactly that data frame, renaming the two dates to **surveillance**'s own defaults so the result can be passed straight through.

With format = "linelist_list" it returns **one line list per stratum** as a tbl_now_surveillance_list, ready to lapply() over – surveillance::nowcast() has no strata argument, so a stratified analysis is one fit per stratum and this saves splitting by hand.

With format = "sts" it instead returns the observed epidemic curve as an surveillance::sts object via surveillance::linelist2sts(), which is what **surveillance**'s plotting and outbreak-detection verbs consume.

now and the delay unit are *not* baked into the result: pass them from the object with get_now() and get_event_units(), as in the example below.

Usage

```
tbl_now_to_surveillance(
  x,
  ...,
  event_col = "dHospital",
  report_col = "dReport",
  format = c("linelist", "linelist_list", "sts"),
  aggregate_by = NULL,
  strata_col = "strata",
  strata_sep = " | ",
  verbose = TRUE
)
```

Arguments

x	A tbl_now object.
...	Forwarded to surveillance::linelist2sts() when format = "sts"; ignored otherwise.
event_col, report_col	Names to give the event and report date columns in the result. Default to **surveillance**'s own "dHospital" and "dReport", so surveillance::nowcast() finds them without further arguments.
format	One of • "linelist" (default) – the single data frame surveillance::nowcast() expects; • "linelist_list" – one line list **per stratum**, as a tbl_now_surveillance_list. Still a plain list, so it goes straight into lapply(); length one and named "all" when the object declares no strata, so the return type does not depend on whether strata happen to be attached; • "sts" – an surveillance::sts object of the observed curve.

aggregate_by	Aggregation interval, e.g. "1 week". NULL (default) derives it from the object's event units ("days" -> "1 day", "weeks" -> "1 week", "months" -> "1 month", "years" -> "1 year"), and aborts on a "numeric" grid, which has integer indices rather than the calendar dates surveillance::linelist2sts() needs. Pass a value explicitly to override, including on a numeric grid if you know what the index steps mean.
strata_col	Name of a single column to add, holding every declared stratum pasted together, for splitting the line list into one fit per stratum. NULL leaves it out. Ignored when the object declares no strata.
strata_sep	Separator used to paste the strata into strata_col.
verbose	Logical. Print the choices that were made.

Value

A data.frame line list (format = "linelist"), a tbl_now_surveillance_list (format = "linelist_list") or an surveillance::sts object (format = "sts").

Stratified nowcasts

surveillance::nowcast() models one series and has no strata argument, so a stratified analysis means fitting each stratum separately. format = "linelist_list" does the splitting, so the fit is an lapply():

```
pieces <- tbl_now_to_surveillance(x, format = "linelist_list", verbose = FALSE)
fits <- lapply(pieces, function(piece) {
  surveillance::nowcast(
    now = max(get_surveillance_range(x)), when = get_surveillance_when(x),
    data = piece, dEventCol = "dHospital", dReportCol = "dReport",
    control = list(dRange = get_surveillance_range(x))
  )
})
```

The control$dRange comes from the **whole object**, not from the piece: every stratum has to be laid on the same time axis, or a stratum whose first case arrived late starts its own time on a different day.

The default format = "linelist" keeps the same information in one frame: the declared strata are pasted into a single strata column, so split(sur, sur$strata) reproduces the list. The original columns are kept alongside it, so you can split on them instead.

Cost of expanding counts

surveillance counts rows, so count-incidence input is expanded to one row per case. **Trim before converting** on a large series: the package's own windowing arguments (when, control$dRange) limit what it *fits*, not what it is handed, and a multi-year daily series can expand into millions of rows.

Censored delays

A censoring indicator that is a property of the **case** rather than of the delay – an administrative "this date is only an upper bound" mark, say – puts a censored and an uncensored row in the same `(event_date, report_date)` cell. A reporting triangle has one slot per cell, so the extra dimension has to go before the conversion. It is removed automatically, with a warning either way:

- **count data**: the counts are summed over the flag, leaving case totals unchanged;
- **line lists**: the column is dropped, leaving one row per case.

`tbl_now_to_epidist()` is the exception and keeps the flag: estimating a delay distribution is the one job that can use it.

See Also

`tbl_now_to_epinowcast()`, tbl_now_surveillance_list

Examples

```
data(denguedat)
nowobj <- tbl_now(denguedat,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)
sur <- tbl_now_to_surveillance(nowobj, verbose = FALSE)
head(sur)

# `now` and the aggregation unit come from the object itself:
get_now(nowobj)
```

---

tbl_now_surveillance_list

One **surveillance** *line list per stratum*

Description

[Stable]

The object returned by `tbl_now_to_surveillance(x, format = "linelist_list")`: one individual-level line list per observed combination of the object's strata, together with the metadata needed to rebuild a `tbl_now` from it.

It is a **thin** class – it is still a list of plain data frames, so `lapply()`, `[[` and friends work as usual:

```
pieces <- tbl_now_to_surveillance(x, format = "linelist_list")
lapply(pieces, function(piece) {
  surveillance::nowcast(
    now = max(get_surveillance_range(x)), when = get_surveillance_when(x),
    data = piece, dEventCol = "dHospital", dReportCol = "dReport",
    control = list(dRange = get_surveillance_range(x))
  )
})
```

The class exists for the same reason tbl_now_triangle_list does: printing says plainly that these are strata rather than something else shaped like a list of line lists, and it carries the now, the units and the original date-column names, none of which survive in a bare `split()`.

now and the time grid are deliberately **not** baked into each piece. The grid must come from the whole object (`get_surveillance_range()`), not from the piece: every stratum has to be laid on the same axis, or a stratum whose first case arrived late starts its own time on a different day.

`as_tbl_now()` binds the pieces back together and restores the original date-column names, the strata and the covariates. Two things do **not** survive, because they are not in the line list to survive: count input comes back as a `"linelist"` (one row per case, so the totals are unchanged but the `case_count` column is gone), and materialised temporal-effect columns come back as ordinary columns rather than as a spec.

Usage

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

Arguments

x	A tbl_now_surveillance_list.
...	Ignored.

Value

`print()` returns x invisibly.

See Also

`tbl_now_to_surveillance()`, `as_tbl_now()`, tbl_now_triangle_list

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat[1:3000, ],
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

## One line list per stratum, in the shape surveillance::nowcast() wants.
linelists <- tbl_now_to_surveillance(dengue,
  format = "linelist_list", verbose = FALSE
)

# Printing summarises the set rather than dumping every data frame.
linelists

length(linelists)
head(linelists[[1]])
```

`tbl_now_triangle_list` *One reporting triangle per stratum*

Description

[Stable]

The object returned by `tbl_now_to_baselinenowcast(x, format = "triangle_list")`: a list of `baselinenowcast::as_reporting_triangle()` matrices, one per observed combination of the object's strata, together with the metadata needed to rebuild a `tbl_now` from it.

It is a **thin** class – it is still a list, so `lapply()`, `[[` and friends work as usual. Use it for **inspecting** per-stratum triangles; for fitting a stratified nowcast, hand the long shape to `baselinenowcast::baselinenowcast()` with its `strata_cols` argument instead – that is the shape it consumes natively, and what `run_nowcast()` does under the hood:

```
long_df <- tbl_now_to_baselinenowcast(x, format = "long")
baselinenowcast::baselinenowcast(long_df, strata_cols = tbl.now::get_strata(x))
```

The class exists for one reason. **baselinenowcast** has a function, `baselinenowcast::estimate_and_apply_delays()`, whose first argument `retro_reporting_triangles` is *also* a list of triangles – but a list of **retrospective** snapshots of one series, used to estimate uncertainty, not one triangle per stratum. Passing this object there would be accepted and would silently treat your strata as successive points in time. Printing the object says plainly what it is, so the mistake is visible rather than silent.

Usage

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

Arguments

`x`	A `tbl_now_triangle_list`.
`...`	Ignored.

Value

`print()` returns `x` invisibly.

See Also

`tbl_now_to_baselinenowcast()`, `as_tbl_now()`

Examples

```
data(denguedat)
dengue <- tbl_now(denguedat[1:3000, ],
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

# One reporting triangle per stratum, in the shape baselinenowcast wants.
triangles <- suppressWarnings(
  tbl_now_to_baselinenowcast(dengue, format = "triangle_list", verbose = FALSE)
)

# Printing summarises the set rather than dumping every matrix.
triangles

# It is a list underneath, so the usual accessors work.
length(triangles)
names(triangles)
```

---

tbl_now_tsibble *Convert between* tbl_now *and* **tsibble**

Description

[Stable]

A tsibble::tsibble() has a single time index and a key identifying each series. Nowcasting needs two time indices, so the conversion keeps both date columns: the index is the **event date** and the report date (plus any strata) becomes part of the key.

tbl_now_from_tsibble() converts a tbl_ts into a tbl_now. You must say which column is the report_date; event_date defaults to the tsibble's index (tsibble::index_var()).

tbl_now_to_tsibble() converts a tbl_now into a tbl_ts, using index ("event_date", the default, or "report_date") as the tsibble index and the other date plus the strata as the key. Linelist data is aggregated to count-incidence first (a tsibble requires unique index/key combinations). The covariates, the censoring indicator and any materialised temporal-effect columns (see compute_temporal_effects()) ride along as measurement columns.

Usage

```
tbl_now_from_tsibble(
  data,
  report_date,
  event_date = NULL,
  strata = NULL,
  ...,
  verbose = TRUE
)
```

```
tbl_now_to_tsibble(
  x,
  ...,
  index = c("event_date", "report_date"),
  verbose = TRUE
)
```

Arguments

data	A tbl_ts (tsibble).
report_date	The report-date column (required for from), as a tidy-select expression – a bare column name or a string.
event_date	The event-date column (for from), as a tidy-select expression – a bare column name or a string. Defaults to the tsibble index.
strata	Optional character vector of strata columns (from). If NULL (default) the tsibble key columns other than the date columns are used.
...	Forwarded to as_tbl_now() (from) or tsibble::as_tsibble() (to).
verbose	Logical. Print the choices that were made.
x	A tbl_now object.
index	For to: which date becomes the tsibble index, "event_date" (default) or "report_date".

Value

A tbl_now (from) or a tbl_ts (to).

Censored delays

A censoring indicator that is a property of the **case** rather than of the delay – an administrative "this date is only an upper bound" mark, say – puts a censored and an uncensored row in the same (event_date, report_date) cell. A reporting triangle has one slot per cell, so the extra dimension has to go before the conversion. It is removed automatically, with a warning either way:

- **count data**: the counts are summed over the flag, leaving case totals unchanged;
- **line lists**: the column is dropped, leaving one row per case.

tbl_now_to_epidist() is the exception and keeps the flag: estimating a delay distribution is the one job that can use it.

See Also

as_tsibble(), the **tsibble** method that calls this; to_count(), since a tsibble needs unique index/key rows and a line list has to be aggregated first; align_weeks() for regular weekly indexes. as_tbl_now() for the generic that dispatches to the *_from_*() side; run_nowcast(), which does the conversion for you when you fit through an engine(). The *One dataset, many nowcasts* article fits the same data with every supported package.

Examples

```
data(denguedat)
nowobj <- tbl_now(denguedat,
  event_date = "onset_week",
  report_date = "report_week", verbose = FALSE
)
# The tsibble is indexed by the event date; the report date is in the key.
ts   <- tbl_now_to_tsibble(nowobj, verbose = FALSE)
back <- tbl_now_from_tsibble(ts, report_date = "report_week", verbose = FALSE)
```

`temporal_effects` *Calendar effects to include in a nowcast*

Description

[Stable]

Reporting follows the calendar. Fewer cases are entered at the weekend, almost none over Christmas, and the backlog clears in the days afterwards. Disease itself follows the calendar too – dengue peaks in the rainy season, influenza in winter. A nowcast that ignores this will overestimate a Monday and underestimate the Tuesday that follows.

`temporal_effects()` writes down which of those patterns you want a model to account for. It does not compute anything: it is a specification you hand to `tbl_now()` or `add_temporal_effects()`, and the columns are only built when `compute_temporal_effects()` is called.

The patterns available are:

- **Position in the week or month** – `day_of_week`, `weekend`, `day_of_month`.
- **Position in the year** – `week_of_year`, `month_of_year`.
- **Smooth seasonality** – `seasons`, which fits Fourier terms rather than one indicator per period. Use this when you believe the pattern is a smooth wave rather than 52 unrelated weeks.
- **Holidays, and the days around them** – `holidays`, `holiday_lags` and `weekend_lags`, which capture the shutdown and the catch-up that follows it.

Usage

```
temporal_effects(
  day_of_week = FALSE,
  weekend = FALSE,
  day_of_month = FALSE,
  month_of_year = FALSE,
  week_of_year = FALSE,
  holiday_lags = 0,
  weekend_lags = 0,
  seasons = integer(0),
  season_length = 1,
  holidays = NULL
)
```

Arguments

day_of_week	Logical. Whether to include an effect for each of the seven days of the week.
weekend	Logical. Whether to include an effect for the weekend vs the weekday.
day_of_month	Logical. Whether to include an effect for the day of the month (1 to 31).
month_of_year	Logical. Whether to include an effect for the month of the year.
week_of_year	Logical. Whether to include an effect for the epidemiological week.
holiday_lags	Single integer (default `0`). Signed depth `N` of the *holiday* lag effect; `holidays` must be supplied whenever `N != 0`. When `N > 0` the effect is placed **after** the holiday: indicator columns `..._holiday_lag_1`, ..., `..._holiday_lag_N` are created, where `..._holiday_lag_k` flags dates that fall exactly `k` **working days** after a holiday. Working days skip weekends (see `weekend_days` in `add_temporal_effects()`) and other holidays, so the effect lands on the first day back at work. Use it to capture a rise in cases just after a holiday. When `N < 0` the effect is placed **before** the holiday instead: columns `..._holiday_lead_1`, ..., `..._holiday_lead_|N|` are created, where `..._holiday_lead_k` flags dates that fall exactly `k` working days *before* a holiday. So `holiday_lags = -1` flags Christmas Eve, and `holiday_lags = -3` flags the last three working days leading up to Christmas.
weekend_lags	Single integer (default `0`). Signed depth `N` of the *weekend* lag effect, mirroring `holiday_lags` but resetting on weekend days rather than holidays (and needing no calendar). When `N > 0`, columns `..._weekend_lag_1`, ..., `..._weekend_lag_N` flag dates exactly `k` working days after a weekend, so with Sat/Sun weekends `weekend_lags = 1` flags the Monday. When `N < 0`, columns `..._weekend_lead_1`, ..., `..._weekend_lead_|N|` flag dates `k` working days before a weekend instead: `weekend_lags = -1` flags the Friday, and `weekend_lags = -3` flags the Wednesday, Thursday and Friday. To model both sides of the same break, add two specifications (see the examples).
seasons	Vector. Smooth (Fourier) seasonality, as opposed to one indicator per calendar unit. Either `integer(0)` (no seasonal effects) or a positive-numeric vector, each entry giving the number of periods in one cycle. The Fourier period of the i-th entry is `seasons[i] * season_length[i]`. In weekly data, `seasons = 52` is a yearly wave. Several entries fit several waves at once, so `seasons = c(7, 365)` in daily data fits a weekly and a yearly cycle together.
season_length	Either a single positive number or a vector of the same length as `seasons`. Specifies the duration (in data units) of each season cycle. Defaults to `1`, meaning the period equals `seasons` directly. Use a value greater than 1 when the data unit is finer than the season. For example, to model 52-week annual seasonality in **daily** data set `seasons = 52, season_length = 7` (period = 364 days).
holidays	Either `NULL` or an `almanac::rcalendar()` specifying how to calculate holidays.

Details

US Federal holidays can be passed by providing the `almanac::cal_us_federal()` calendar.

Example:

```
library(almanac)
temporal_effects(holidays = cal_us_federal())
```

Value

An object of class `temporal_effects`: a specification, not data. Hand it to `tbl_now()`'s `t_effects` argument or to `add_temporal_effects()`.

Using a different holiday calendar

`holidays` accepts **any** `almanac::rcalendar()`.

A calendar is a set of *recurrence rules*: you describe how a holiday is constructed – "the fourth Thursday of November" – and **almanac** generates it for every year. Avoid hardcoding specific dates such as `"2021-11-18"`, which will be wrong next year.

A calendar has four building blocks:

- **Built-in holidays. almanac** ships rules for common US holidays: `hol_us_thanksgiving()`, `hol_us_memorial_day()`, `hol_christmas()`, `hol_us_election_day()`, and so on. See `almanac::rholiday()` for the list.
- **Your own holidays.** Anything without a built-in rule is a `yearly()` recurrence narrowed with `recur_on_*()` and named with `almanac::rholiday()`.
- **Observance.** `almanac::hol_observe()` shifts a fixed-date holiday that lands on a weekend onto a working day. `adjustment = adj_nearest` gives the usual US rule (Saturday moves back to Friday, Sunday forward to Monday); `adj_following` and `adj_preceding` always move one way.
- **Editing a calendar.** `almanac::cal_add()` and `almanac::cal_remove()` tweak an existing calendar, and `almanac::cal_names()` lists what is in one.

Use `almanac::cal_events()` to check what you built before modelling with it.

Worked example: the New York City calendar:

NYC observes the US federal holidays plus Lincoln's Birthday and Election Day, and calls the October holiday Columbus Day. Only Lincoln's Birthday needs a hand-written rule; everything else is built-in, with `hol_observe()` on the fixed-date holidays.

```
library(almanac)

cal_nyc <- function(since = NULL, until = NULL) {

  #Adjust if a holiday happens on a weekend move to the closest date
  #i.e. 4th of July on Saturday in 2026 moves to Friday July 3rd
  on_weekends <- recur_on_weekends(weekly(since = since, until = until))
  observed <- function(x) {
    hol_observe(x, adjust_on = on_weekends, adjustment = adj_nearest)
```

```
  }

  # Build a rule for Lincoln's birthday: February 12th, every year.
  lincolns_birthday <- yearly(since = since, until = until) |>
    recur_on_month_of_year("February") |>
    recur_on_day_of_month(12L) |>
    rholiday(name = "Lincoln's Birthday")

  rcalendar(
    #New years day moves to closest weekday
    observed(hol_new_years_day(since = since, until = until)),
    #MLK day happens that day
    hol_us_martin_luther_king_junior_day(since = since, until = until),
    #Lincoln's birthday moves to closest weekday
    observed(lincolns_birthday),
    #President's day happens that day
    hol_us_presidents_day(since = since, until = until),
    #Memorials day happens that day
    hol_us_memorial_day(since = since, until = until),
    #Juneteenth is moved to closest weekday
    observed(hol_us_juneteenth(since = since, until = until)),
    #4th of July is moved to closest weekday
    observed(hol_us_independence_day(since = since, until = until)),
    #Labor day happens that specific day
    hol_us_labor_day(since = since, until = until),
    #We can rename what almanac names Indigenous People's day to Columbus
    hol_rename(
      hol_us_indigenous_peoples_day(since = since, until = until),
      "Columbus Day"
    ),
    #Election day
    hol_us_election_day(since = since, until = until),
    #Veteran's day moves closest
    observed(hol_us_veterans_day(since = since, until = until)),
    #Thanksgiving happens that specific Thursday
    hol_us_thanksgiving(since = since, until = until),
    #Christmas moves to closest day
    observed(hol_christmas(since = since, until = until))
  )
}

# Check it before using it. The same rules generate any year you ask for:
cal_events(cal_nyc(), year = 2026, observed = TRUE)
cal_events(cal_nyc(), year = 2027, observed = TRUE)

# Then hand it to temporal_effects() like any other calendar:
temporal_effects(holidays = cal_nyc())
```

Two of those show the rules we implemented:

- **Independence Day is Jul 3, not Jul 4.** Jul 4 2026 is a Saturday, so `adj_nearest` moves the observance *back* to Friday Jul 3. In 2027 it lands on a Sunday and moves *forward* to Mon Jul 5.
- **Christmas 2027 is observed on Fri Dec 24**, and New Year's Day 2028 is pulled back to Fri Dec 31 2027 — so it appears in the 2027 events, not 2028.

See Also

`add_temporal_effects()` to attach a specification to a `tbl_now`, and `compute_temporal_effects()` to turn it into columns; get_temporal_effects() to read it back; calendar_effect_plots to see the patterns in your data before deciding which to model; `is_weekday()` for the weekend definition these use; `almanac::rcalendar()` for building holiday calendars.

Examples

```
temporal_effects(day_of_week = TRUE, week_of_year = TRUE)

## Annual seasonality in weekly data (period = 52 weeks)
temporal_effects(seasons = 52)

## Annual seasonality in daily data (52 weeks x 7 days = 364-day period)
temporal_effects(seasons = 52, season_length = 7)

# After-weekend effect: flag the first two working days after a weekend
temporal_effects(weekend = TRUE, weekend_lags = 2)

## Before-weekend effect: flag the last working day before a weekend (Friday)
temporal_effects(weekend = TRUE, weekend_lags = -1)

if (rlang::is_installed("almanac")) {
  cal <- almanac::rcalendar(almanac::hol_christmas())
  temporal_effects(holidays = cal, day_of_month = TRUE, seasons = c(7, 365))

  # After-holiday effect: flag the first 3 working days back after a holiday
  temporal_effects(holidays = cal, holiday_lags = 3)

  # Before-holiday effect: flag the 2 working days leading up to a holiday
  temporal_effects(holidays = cal, holiday_lags = -2)

  # A calendar of your own: write a rule for the holiday, not a date, and
  ## almanac generates it for every year (see "Using a different holiday
  # calendar" above for a full local calendar).
  lincolns_birthday <- almanac::yearly() |>
    almanac::recur_on_month_of_year("February") |>
    almanac::recur_on_day_of_month(12L) |>
    almanac::rholiday(name = "Lincoln's Birthday")

  # Add it to the federal calendar and check what you built
  cal_local <- almanac::cal_add(almanac::cal_us_federal(), lincolns_birthday)
  almanac::cal_events(cal_local, year = 2026, observed = TRUE)

  temporal_effects(holidays = cal_local, holiday_lags = 2)
```

```
  # Both sides of the holiday: add one specification per direction
  data(denguedat)
  tbl_now(denguedat,
    event_date = onset_week, report_date = report_week, verbose = FALSE
  ) |>
    add_temporal_effects(temporal_effects(holidays = cal, holiday_lags = -2)) |>
    add_temporal_effects(temporal_effects(holidays = cal, holiday_lags = 2))
}
```

`tidy.delay_distribution` *Tidy a fitted delay distribution*

Description

[Stable]

epidist and `EpiNow2::estimate_dist()` (new in **EpiNow2** 1.9.0) do not produce a nowcast: they estimate a **reporting-delay distribution**. There are therefore no per-event-date case estimates to tidy, and the columns `tidy.nowcast()` promises (`event_date`, `stratum`, ...) would all be meaningless. Both methods instead return the same *delay-shaped* table: one row per distribution parameter, with `term` rather than `event_date`, so the two packages' fits can be compared side by side.

Usage

```
## S3 method for class 'estimate_dist'
tidy(x, probs = NULL, level = 0.95, ...)

## S3 method for class 'epidist_fit'
tidy(x, probs = NULL, level = 0.95, newdata = NULL, ...)
```

Arguments

`x`	A fit from `epidist::epidist()` or `EpiNow2::estimate_dist()`.
`probs`	Optional numeric vector of probabilities in `[0, 1]`, adding one `q*` column each.
`level`	Width of the reported interval. Defaults to `0.95`.
`...`	Unused, for generic consistency.
`newdata`	`tidy.epidist_fit()` only. Optional data frame passed to `epidist::predict_delay_parameters()`, for a fit with covariates in the delay model (`formula = mu ~ 1 + gender`, say). `NULL` uses the fit's own data.

Value

A tibble, as described in *Value*.

Value

A tibble with one row per parameter of the fitted delay distribution – whichever family was fitted, named as the fitting package names it – plus the derived `mean` and `sd` of the delay, which are the numbers most people actually want. The columns are:

`term character`. The parameter: the distribution's own parameters (`mu`, `sigma`, ...) plus `mean` and `sd`.

`estimate numeric`. Posterior median.

`conf.low`, `conf.high numeric`. Interval bounds, following **broom**'s naming.

`level numeric`. The width of that interval.

`engine character`. `"epidist"` or `"EpiNow2"`.

Everything is summarised from the posterior draws, so `level` is the interval you asked for rather than whichever `CrIs` the fit happened to use, and `probs` appends one `q*` column per requested probability – a real quantile rather than an approximation.

How `mean` and `sd` are obtained

epidist reports continuous-distribution moments via `epidist::add_mean_sd()`.

EpiNow2 gets them without naming a distribution. It can fit five families today and may add more, and a `switch()` in this package would quietly stop reporting anything the day it does. Instead each draw's parameters are put back into the fit's own `dist_spec` and discretised with `EpiNow2::discretise()`, which knows the families; the moments are then a summation over the PMF. A new family works as soon as `discretise()` supports it.

The trade-off is that the **EpiNow2** numbers are the moments of the **discretised** delay – the distribution **EpiNow2** convolves with downstream. Against the closed forms the mean is exact and the sd runs about 1% high, that being the variance a discrete grid adds. Expect a difference of that order when comparing the two packages' fits.

A name collision worth knowing about

`summary()` on an `estimate_dist` fit has `mean` and `sd` **columns**, and those are the posterior mean and sd **of the parameter** on that row – not of the delay. The `mean` and `sd` these methods report are **rows**, and are the delay distribution's own moments. Same words, different quantities.

Dispatch

`epidist()` returns an object of class `c("brmsfit", "epidist_fit")`, in that order, so if **broom.mixed** is loaded its `tidy.brmsfit()` method matches **first** and you get raw **brms** parameters instead of this table. Call `tidy.epidist_fit(fit)` explicitly when you want the delay distribution and cannot be sure which method will win.

See Also

`tbl_now_to_epidist()` and `tbl_now_to_EpiNow2()` for the conversions; tidy() for tidying a *case-count* nowcast rather than a delay distribution; revision_delay for the delay these are estimating.

Examples

```
data(denguedat)
# A short window: fitting a delay distribution does not need twenty years of
# data, and Stan is slow.
recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
nowobj <- tbl_now(recent,
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
)

# `target = "estimate_dist"` gives the censored linelist EpiNow2 wants: one
# row per case, each date as the interval it is known to fall in.
delays <- tbl_now_to_EpiNow2(nowobj,
  target = "estimate_dist", verbose = FALSE, quiet = TRUE
)
head(delays)

# A short chain keeps the example quick -- use EpiNow2's defaults for real
## work. `try()` guards the case where EpiNow2 is installed but its Stan
# toolchain is not.
fit <- try(
  EpiNow2::estimate_dist(
    delays,
    stan = EpiNow2::stan_opts(samples = 100, chains = 1)
  ),
  silent = TRUE
)

# One row per fitted parameter, plus the delay's own mean and sd.
if (!inherits(fit, "try-error")) {
  print(tidy(fit))
  print(tidy(fit, probs = c(0.05, 0.95)))
}
```

`tidy.nowcast` *Tidy a fitted nowcast into one standard table*

Description

[Stable]

Every nowcasting package returns its answer in its own shape – a matrix of posterior draws, an `stsNC` object, a Stan fit, an INLA summary, a bare list. `tidy()` turns any of them into the **same** table, so downstream code (plotting, scoring, comparison) does not care which engine produced it.

Usage

```
tidy(x, ...)

## S3 method for class 'baselinenowcast_df'
```

```
tidy(x, probs = NULL, ...)

## S3 method for class 'epinowcast'
tidy(x, probs = NULL, ...)

## S3 method for class 'stsNC'
tidy(x, probs = NULL, ...)

## S3 method for class 'estimate_infections'
tidy(x, probs = NULL, ...)

## S3 method for class 'epinow'
tidy(x, probs = NULL, ...)

## S3 method for class 'estimate_truncation'
tidy(x, probs = NULL, ...)

## S3 method for class 'list'
tidy(x, probs = NULL, engine = NULL, level = NULL, ...)
```

Arguments

x	A fitted nowcast. See *Supported objects*.
...	Passed to methods.
probs	Optional numeric vector of probabilities in `[0, 1]`. Adds a `q*` column per probability. Only available for engines that expose draws.
engine	Optional string naming the engine. Needed only for the shapes that arrive as an **unclassed list** – a **NobBS** fit, an `EpiNow2::regional_epinow()` result, or a per-stratum list of `surveillance::nowcast()` fits – which are otherwise recognised by their structure.
level	Interval width to report for an engine that does not say what it produced. Only used by the **NobBS** branch (see `level` under *Value*); `NULL`, the default, reports `NA`.

Value

A tibble, as described above.

Value

A tibble with one row per event date (per stratum, where the fit carries strata) and these columns:

`event_date Date`. The event/reference date, **on the engine's own grid**. `tidy()` deliberately does not re-grid: some packages bin onto week starts of their own choosing, and silently snapping them would hide a real difference. Align afterwards if you need to.

`stratum character`. One label per stratum the fit reports, and `"all"` when the fit is unstratified. Several stratifying columns are pasted `" | "`-separated, matching the `triangle_list` naming of `tbl_now_to_baselinenowcast()`. `(stratum, event_date)` is therefore a unique key.

`estimate` `numeric`. The point nowcast – the posterior median where the engine provides draws or a median, otherwise its point estimate.

`conf.low`, `conf.high` `numeric`. Interval bounds, following **broom**'s naming. `NA` when the engine returns no interval.

`level` `numeric`. The width the interval actually has, e.g. `0.95`. Engines differ – **epinowcast** reports a 90% band by default while others report 95% – and without this column those get compared as if they were the same thing. `NA` whenever the width cannot be established: because the engine returned no interval (a **baselinenowcast** fit made with `output_type = "point"`), or because it returned one without saying how wide it is. **NobBS** is the latter case – its `lower`/`upper` come from `specs$conf`, and `NobBS()` does not return `specs` – so pass `level` yourself if you need it filled in. A guessed default is worse than `NA` in the one column that exists to stop widths being compared blindly.

`engine` `character`. Which package produced the fit.

When `probs` is supplied, one extra column per requested quantile is appended, named `q5`, `q50`, `q95` and so on (the probability times 100, so `0.025` becomes `q2.5`).

Which engines can honour `probs`

Only the engines that expose **draws** can compute an arbitrary quantile: **diseasenowcasting**, **baselinenowcast** and **epinowcast**. The others report a fixed set of summaries and nothing else, so asking them for a quantile they did not compute is an error rather than a silent approximation.

Supported objects

- `nowcast_prediction` (S7) from `diseasenowcasting::predict()`
- `baselinenowcast_df` from `baselinenowcast::baselinenowcast()` – both the single-series fit (from a `reporting_triangle`) and the stratified fit (from a long `data.frame` with `strata_cols = `). A stratified fit is tidied one row per (stratum, event_date), with the strata columns pasted into the `stratum` label.
- `epinowcast` fits
- `stsNC` from `surveillance::nowcast()`
- the list returned by `NobBS::NobBS()` or by `NobBS::NobBS.strat()` (the stratified variant is recognised by its `stratum` column)

See Also

`run_nowcast()` and tidy(), which give you this shape without needing to call the modelling package yourself; tidy() for a backtest; `tidy.epidist_fit()` and `tidy.estimate_dist()` for fitted *delay distributions* rather than case counts; `score_nowcast()` to score the result. The *One dataset, many nowcasts* article shows each engine's native output next to this one.

Examples

```
data(denguedat)
# A few years of data and a small number of draws, to keep the example quick.
dengue <- tbl_now(denguedat[1:10000, ],
  event_date = "onset_week", report_date = "report_week", verbose = FALSE
```

```
)
triangle <- suppressWarnings(
  tbl_now_to_baselinenowcast(dengue, verbose = FALSE)
)
fit <- baselinenowcast::baselinenowcast(
  triangle, output_type = "samples", draws = 25
)
tidy(fit)
```

`tidy.nowcast_backtest` *Tidy the predictions and scores of a* `nowcast_backtest()`

Description

[Stable]

One row per (method, now date, target) carrying both halves of the comparison – what the model said and what happened – with the dot-prefixed internal column names traded for ordinary ones so the result goes straight into **dplyr** or **ggplot2**.

Usage

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

Arguments

`x`	A `nowcast_backtest` object.
`...`	Unused, for generic consistency.

Value

A tibble with the columns `method`, `now`, `event_date`, `stratum`, `observed`, `estimate`, `conf.low`, `conf.high`, `level`, `wis`, `ae_median`, `coverage_50` and `coverage_90`. `stratum` is `"all"` for an unstratified backtest and the `" | "`-pasted strata otherwise, so `(method, now, stratum, event_date)` is a unique key.

`estimate`, `conf.low`, `conf.high` and `level` are the retrospective prediction itself, read off the same quantiles the scores were computed from and named as tidy() names them: `estimate` is the `0.5` quantile and `level` the width of the **widest symmetric pair actually present**. `nowcast_backtest()` refuses engines that report different quantile levels, so `level` is one number for the whole table. When no symmetric pair exists all three of `conf.low`, `conf.high` and `level` are `NA`, and `estimate` is `NA` when the median was not among the levels reported – a guessed width defeats the point of the column.

See Also

`nowcast_backtest()`, which produces the object being tidied; `nowcast_weights()` to turn the same scores into ensemble weights; `score_nowcast()` for scoring a single nowcast; tidy() for a fitted nowcast rather than a backtest.

Examples

```
data(denguedat)
recent <- subset(denguedat, onset_week >= as.Date("2010-06-01"))
dengue <- tbl_now(recent,
  event_date = onset_week, report_date = report_week, verbose = FALSE
)

## `example_engine()` is a toy that ignores the reporting delay entirely; it
# is used here only so the example runs without a modelling package.
## Swap in a real one -- `engine_baselinenowcast()`, `engine_epinowcast()`,
## `engine_nobbs()` -- for anything you intend to act on.

bt <- nowcast_backtest(dengue,
  example_engine(label = "carry forward"),
  now_dates = as.Date(c("2010-10-04", "2010-11-15")), verbose = FALSE
)

# One tidy row per method, `now` date, stratum and event date, carrying the
# retrospective prediction next to the resolved truth used for scoring.
head(tidy(bt))
```

---

tidy.tbl_nowcast *Tidy a nowcast produced by* run_nowcast() *or* nowcast_ensemble()

Description

[Stable]

Turns a tbl_nowcast into the same table every other `tidy()` method in this package returns, so a nowcast produced through `run_nowcast()` and one produced by calling a modelling package by hand are read the same way.

Usage

```
## S3 method for class 'tbl_nowcast'
tidy(x, probs = NULL, ...)
```

Arguments

x	A tbl_nowcast.
probs	Optional numeric vector of probabilities in `[0, 1]`, adding one `q*` column each. Only available when the nowcast carries **draws**: a quantile-only nowcast cannot produce a level it was not summarised at, so asking for one is an error rather than an interpolation dressed up as a quantile.
...	Unused, for generic consistency.

Details

Registered by hand in `.onLoad()`. The S7 class name is `tbl.now::tbl_nowcast`, so the S3 method dispatch actually looks up is `tidy.tbl.now::tbl_nowcast` – not a writable R name, and beyond what `@exportS3Method` can express. The function is nonetheless *named* for the method it implements, because `R CMD check` resolves this topic's usage section back to an object of that name; a helper called something else would leave the help page documenting a function that does not exist.

Value

A tibble, as described in *Value*.

Value

A tibble with the columns documented at `tidy.nowcast()`: `event_date`, `stratum`, `estimate`, `conf.low`, `conf.high`, `level` and `engine`, plus one `q*` column per element of `probs`.

Two of those columns are read off the object rather than assumed:

`level` A `tbl_nowcast` holds whatever quantile levels it was summarised at, and those need not be symmetric. `level` is the width of the **widest symmetric pair actually present** – `0.95` for the default `nowcast_quantile_levels()`, `0.8` for a fit summarised at `c(0.1, 0.5, 0.9)`. When no symmetric pair exists, `level`, `conf.low` and `conf.high` are all `NA`: a guessed width defeats the point of the column.

`engine` The nowcast's `method`, so `"baselinenowcast"` for a single fit and `"ensemble"` (or whatever `name` `nowcast_ensemble()` was given) for a combined one.

`stratum` is `"all"` only when the nowcast declares no strata. Several strata columns are pasted `" | "`-separated, matching the rest of the package, so `(stratum, event_date)` is a unique key.

`estimate` is the `0.5` quantile, and `NA` when the nowcast was summarised at levels that do not include the median.

See Also

`tidy.nowcast()` for the same table off a raw engine fit, `run_nowcast()`, `nowcast_ensemble()`.

Examples

```
predictions <- tidyr::expand_grid(
  onset_week = as.Date("2020-01-05") + c(0, 7),
  .quantile_level = c(0.025, 0.5, 0.975)
)
predictions$.value <- c(5, 10, 18, 6, 12, 21)
nc <- tbl_nowcast(
  predictions = predictions, method = "toy", event_date = "onset_week"
)

tidy(nc)
```

`to_count` *Convert between linelist and aggregated count data*

Description

[Stable]

Surveillance data comes in three shapes, and different nowcasting packages want different ones. `to_count()` moves a `tbl_now()` between them:

- `linelist` – one row per case. The most detailed shape.
- `count-incidence` – one row per (event date, report date) pair, holding the number of cases reported *on exactly that report date*.
- `count-cumulative` – the same grid, but holding the number of cases known *up to and including* that report date. This is the shape most public dashboards publish.

You can go from `linelist` to either count shape, and back and forth between the two count shapes. You cannot go back to `linelist`: once cases have been added up, the individual rows are gone.

Usage

```
to_count(x, to = NULL, ...)

## S3 method for class 'tbl_now'
to_count(x, to = NULL, ...)
```

Arguments

`x`	A `tbl_now()` object to convert.
`to`	Character. The data type to produce: `"linelist"`, `"count-incidence"` or `"count-cumulative"`. Defaults to the object's current type, i.e. no change.
`...`	Additional arguments passed to methods.

Details

This is an S3 generic. The package provides a method for `tbl_now` objects, which aggregates into the `case_count` column, creating one named `n` when the object does not already have one.

Aggregation sums over every column the object has *not* been told about, so a column you care about should be declared as a strata or covariate first (see `add_strata()`) or it will be summed away.

Value

A `tbl_now` object of the requested `to` data type, with the counts aggregated into the `case_count` column.

Grouping is dropped, and said so

`to_count()` **ungroups**, and warns when it does. It is the one verb in the package that does not put the caller's grouping back, and the reason is that it changes what a row *is*: after aggregating, one row is an (event date, report date) cell rather than one of the rows that were grouped, so the grouping no longer describes anything in the object.

A grouping is also not how you keep a column out of the sum. Declare it – `add_strata()` or `add_covariates()` – and it becomes part of the cell key. The reported-cases getters do respect a grouping, because they select rather than reshape; see `get_latest_reported_cases()`.

Statistical details

Converting `count-cumulative` to `count-incidence` **de-accumulates** the series: within each event date (and grouping), ordered by report date, the increment is that cumulative total minus the previous one. Because published cumulative totals are sometimes revised *downward*, an increment can be **negative**. That is not a bug – it is a retraction showing through – but code that requires non-negative counts (for example `tbl_now_to_baselinenowcast()`) must handle or refuse it.

Note

`linelist` data cannot be reconstructed from `count-*` data. Asking for it throws an error, because aggregated data cannot be un-counted.

See Also

`tbl_now()` and its *Data types* section for what each shape means; `get_data_type()` to ask an object which shape it currently is; `complete_zeroes()` to fill in the (event, report) pairs where nothing was reported; `get_latest_reported_cases()` to pull out the most recent counts.

Examples

```
data(denguedat)
ndata <- tbl_now(denguedat,
  event_date = "onset_week",
  report_date = "report_week",
  strata = "gender"
)

# A linelist has one row per case ...
nrow(ndata)

## ... which becomes one row per (onset week, report week, gender), with the
# number of cases in `n`.
counts <- to_count(ndata, to = "count-incidence")
counts

# Cumulative totals instead: how many cases for that onset week were known by
# each report week. Within an onset week these only ever go up.
to_count(counts, to = "count-cumulative")

# Going back to a linelist is impossible -- the individual cases are gone.
```

```
try(to_count(counts, to = "linelist"))
```

`transport_discriminant` *The transport discriminant of a reporting series*

Description

[Experimental]

Computes, for every report date, the two coordinates of `diagnose_batches()`'s conservation law – the **deficit** (the *transport* axis: how many reports the preceding window is missing) and the window **discriminant** (the *creation* axis: the window total relative to its baseline) – together with their robust standardised versions `transport_z` and `creation_z`.

Usage

```
transport_discriminant(
  x,
  lookback = 7L,
  baseline_window = NULL,
  period = NULL,
  alpha = 0.05,
  axis = c("report", "revision"),
  drop_censored = TRUE
)
```

Arguments

`x`	A `tbl_now()` object.
`lookback`	Integer window half-width k (report-grid steps) over which the deficit is accumulated. Default 7 (a week of daily reporting).
`baseline_window, period`	Baseline controls, passed through to the same machinery as `diagnose_batches()`. `period` (e.g. 7) absorbs a scheduled weekly reporting cadence.
`alpha`	Level for the `classification` labels. Default `0.05`.
`axis`	Which time axis to scan for arrivals: `"report"` (default) or `"revision"`. Needs a revision process (see `add_revision_date()`); cases still `"pending"` are left out.
`drop_censored`	Logical. Ignore the rows whose date on `axis` is flagged censored (`is_censored_report`, or `is_censored_revision` on the revision axis). Default `TRUE`: a censored date is a *bound*, not the date the record arrived, so those rows would pile up on the censoring date and be rediscovered as the very batch the censoring already recorded.

Details

A batch *moves* reports later without creating them, so it leaves a positive deficit while conserving the window total (`transport_z` large, `creation_z` near 0). A genuine surge *creates* reports, lifting the window total without a deficit (`creation_z` large, `transport_z` near 0). Reading the two together separates a backlog release from an epidemic surge: a point sits in the **batch** corner when its transport score is large and its creation score is not. A negative `creation_z` with no transport is a hold in progress (the window is depleted and nothing has been released yet). The `classification` column applies these labels at level `alpha`, exactly as in `diagnose_batches()`.

Value

A tibble of class `transport_discriminant`, one row per (report date, stratum), with columns `report_date`, `stratum`, `reported`, `baseline`, `window_total`, `spike` (reported minus baseline), `deficit`, `delta`, `transport_z`, `creation_z`, `classification` and `batch`.

See Also

`diagnose_batches()` for the hypothesis test, `diagnostic_plot()` to plot this plane.

Examples

```
data(denguedat)
# The two and a half years around the 1996 and 1997 backlog dumps. The whole
# twenty-year series works the same way, it just takes longer to scan.
window <- denguedat[
  denguedat$onset_week >= as.Date("1995-06-01") &
    denguedat$onset_week <= as.Date("1998-01-01"),
]
dn <- tbl_now(window, onset_week, report_week, verbose = FALSE)
td <- transport_discriminant(dn)
td[td$batch, ]
```

`update.tbl_now` *Append newly arrived data to a* `tbl_now`

Description

[Experimental]

Surveillance data does not arrive once; it arrives every week. `update()` takes a `tbl_now` and a batch of newer rows – as another `tbl_now` or as a plain `data.frame` – and returns a single object containing both, still knowing everything the original knew about itself.

It also moves `now` forward, because the new rows may carry a later report than the object had seen. That is the difference between this and `dplyr::bind_rows()`, which would give you back a plain data frame with no idea what a nowcast is.

Usage

```
## S3 method for class 'tbl_now'
update(
  object,
  ...,
  new_data,
  strata = "left",
  covariates = strata,
  t_effects = strata,
  now = NULL,
  remove_duplicates = NULL
)
```

Arguments

object	A tbl_now object
...	Additional arguments to pass to tbl_now
new_data	Another tbl_now with the same strata, covariates, is_censored_report, and temporal_effects or a data.frame with additional (newer) data not present in x
strata	(optional) Whether to keep the strata from object ("left"), from new_data ("right") or from both ("both")
covariates	(optional) Whether to keep the covariates from object ("left"), from new_data ("right") or from both ("both")
t_effects	(optional) Which temporal-effects spec to keep: from object ("left", the default), from new_data ("right") or the union of both ("both").
now	(optional) Date or NULL (default). The date that is considered the now of the nowcast. If no now is given then the function automatically uses the last event_date.
remove_duplicates	Whether to remove duplicated rows from data (only applies for count data)

Value

A tbl_now object with all the properties of object

Note

By default it keeps the strata, covariates and temporal effects of object. Use the strata, covariates and t_effects arguments to change it.

See Also

update_now() to move now without adding rows; tbl_now() for the attributes that are carried over; add() and change() to edit those attributes instead of the data.

Examples

```
data(denguedat)

# Pretend the first 500 rows are what you had last week ...
initial_tbl <- tbl_now(denguedat[1:500, ],
  event_date = "onset_week",
  report_date = "report_week", strata = "gender",
  verbose = FALSE
)
nrow(initial_tbl)
get_now(initial_tbl)

# ... and these arrived since.
new_rows <- denguedat[501:1000, ]

# The result has both, keeps `gender` as a stratum, and has moved `now`
# forward to the latest report it has now seen.
updated <- update(initial_tbl, new_data = new_rows)
nrow(updated)
get_strata(updated)
get_now(updated)
```

`validate_tbl_now` *Check that an object is a valid* `tbl_now`

Description

[Stable]

Two different questions about an object, and one function for each.

- `is_tbl_now()` asks **"is this the class?"**. It answers quietly with `TRUE` or `FALSE`, and it is cheap: a class check, the attributes a `tbl_now` cannot do without, and the columns those attributes name. Use it in an `if`.
- `validate_tbl_now()` asks **"is the data in it sane?"**. It answers loudly: it stops with an error explaining what is wrong, and warns about the merely suspicious. Use it when you want the pipeline to halt rather than carry on with a broken object.

Neither checks whether the data are *good* – only whether the object is put together correctly. For the quality of the data itself, use `diagnose()`.

Usage

```
validate_tbl_now(x, warn_non_uniqueness = FALSE, warn_now = TRUE)

is_tbl_now(x)
```

Arguments

x	An object to check.
warn_non_uniqueness	(optional) Logical. Whether to throw a warning if data has several rows on the same full key: the event, report and (when declared) revision dates, the revision type, the strata, the covariates and the censoring flags. Rows carrying an NA anywhere in that key – a pending case has no revision date – are left out of the comparison, because an unobserved value cannot be shown to equal another.
warn_now	Boolean. Whether to warn if now falls before the last report date, or unreasonably far into the future.

Details

validate_tbl_now() and diagnose() share one implementation. This function is the *condition* presentation of it: it aborts on the error findings and warns about the warning ones. diagnose() is the *data* presentation, and additionally reports the note-level observations that would make every dplyr verb noisy if they were emitted here.

is_tbl_now() deliberately runs **none** of that. It used to, and the cost was paid twice over: the findings engine ran on every .assert_tbl_now(), and the warnings it raised escaped – so an object the user had already chosen to keep re-reported its problems from wherever the predicate happened to be called. An object can therefore be a tbl_now (is_tbl_now() is TRUE) and still have data validate_tbl_now() warns about. That is the point: the class is a container, and a container is not a claim that what is in it is clean.

Value

is_tbl_now() returns a single TRUE or FALSE.

validate_tbl_now() returns TRUE invisibly; it is called for the error or warning it raises when the object is malformed.

See Also

diagnose() for the same findings returned as a tibble, plus the softer notes; tbl_now() to build a valid object; tbl_now_attributes() to see what it recorded. The *Diagnosing a tbl_now* article explains what each finding means.

Examples

```
data(denguedat)
ndata <- tbl_now(denguedat,
  event_date = "onset_week",
  report_date = "report_week", verbose = FALSE
)

# A well-formed object passes both checks.
is_tbl_now(ndata)
validate_tbl_now(ndata)

# `is_tbl_now` is a question about the CLASS, so it stays quiet about the
```

```
# data. This object's report dates include an `NA`, which validate_tbl_now
# warns about -- and which does not stop it being a `tbl_now`.
messy <- ndata
messy$report_week[1] <- NA
is_tbl_now(messy)

# A plain data.frame is not a tbl_now ...
is_tbl_now(data.frame(x = 1:3))

## ... and asking for revision says so, with a reason. The call below is
# wrapped in `try` because it is meant to fail here.
try(validate_tbl_now(data.frame(x = 1:3)))
```

Index