Package 'spatialkit'

September 11, 2026

Title Spatial Tessellation, Modeling, and Cross-Validation Toolkit

Version 2.0.0

Description Constructs analysis regions from the distribution of the data itself, as an alternative to aggregating onto administrative boundaries that were drawn for unrelated purposes. Seeds and builds Voronoi, Delaunay, hexagonal and square tessellations with reproducible identifiers, selects a cell count from the spatial structure of the observations, assigns features to cells, and aggregates to cell level with optional design-effect corrections so that standard errors account for within-cell autocorrelation. Also manages coordinate reference systems. Fits geographically weighted regression (via 'GWmodel'; Lu et al. (2014) <doi:10.1080/10095020.2014.917453>), Bayesian spatial Gaussian process regression (via 'brms', using the Hilbert space approximation of Riutort-Mayol et al. (2023) <doi:10.1007/s11222-022-10167-2>) and random forests (via 'ranger', with the permutation importance of Strobl et al. (2007) <doi:10.1186/1471-2105-8-25>), each behind one S3 class with consistent predict, fitted, residuals and plot methods. Provides spatial cross-validation with random, block, buffered, leave-location-out and nearest-neighbour distance-matched folds (Mila et al. (2022) <doi:10.1111/2041-210X.13851>), forward variable selection, model comparison, prediction onto a regular surface, and the area of applicability of Meyer and Pebesma (2021) <doi:10.1111/2041-210X.13650> to flag where a fitted model extrapolates beyond its training data.

License MIT + file LICENSE

URL https://github.com/elkronos/gis_modeling_toolkit

BugReports https://github.com/elkronos/gis_modeling_toolkit/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports sf (>= 1.0), dplyr (>= 1.0), logger, digest, stats, methods, utils, parallel

Suggests sp, GWmodel, ranger, brms (>= 2.17.0), cmdstanr, loo, tibble, geometry, gstat, ggplot2, patchwork, FNN, Matrix, spdep, testthat (>= 3.1.5), knitr, rmarkdown

Additional_repositories https://stan-dev.r-universe.dev

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Justin Chase [aut, cre, cph]

Maintainer Justin Chase <jchase.msu@gmail.com>

Repository CRAN

Date/Publication 2026-09-11 07:00:02 UTC

Contents

`area_of_applicability` *Area of applicability of a spatial prediction model*

Description

Computes the dissimilarity index (DI) of Meyer & Pebesma (2021) for a set of prediction locations and flags those that fall inside the model's area of applicability (AOA) – the region of predictor space where the model's cross-validated performance estimate can be expected to hold.

Usage

```
area_of_applicability(
  newdata,
  model = NULL,
  train_sf = NULL,
  predictor_vars = NULL,
  weights = NULL,
  folds = NULL,
```

```
  threshold = NULL,
  normalizer_max_n = 5000L,
  seed = 123L,
  chunk_size = NULL,
  use_fnn = requireNamespace("FNN", quietly = TRUE)
)
```

Arguments

newdata	Prediction locations: an sf object (typically from predict_surface) or a data.frame, carrying the predictor columns. For a model fitted with include_coords = TRUE an sf object is required, non-POINT geometry is reduced to representative points, and a CRS mismatch with the training data is reconciled; see *Models fitted with the coordinates as predictors*.
model	A fitted spatial_fit, supplying the training data and predictor names. Optional if train_sf and predictor_vars are given directly, which lets this be used with any model.
train_sf	Training data, if not taken from model.
predictor_vars	Predictor names, if not taken from model.
weights	Optional named numeric vector of predictor importances. Any positive scale works. When model was fitted with include_coords = TRUE, the coordinates ".x" and ".y" are measured as predictors too (see below), and you are not expected to supply importances for them: any you leave out default to the mean of the weights you did supply, so location counts about as much as a typical predictor. Naming them explicitly overrides that. An unnamed vector may have one value per predictor either with or without the two coordinate columns.
folds	Cross-validation folds: a make_folds result, a list of train/test splits, or a vector of fold labels with one entry per training row. Default NULL (plain nearest neighbour).
threshold	Optional numeric override for the DI threshold.
normalizer_max_n	Subsample the training data to this many points when computing the mean pairwise distance, which is quadratic. Default 5000.
seed	Seed for that subsample. Default 123.
chunk_size	Query rows per distance block on the dense path. Default NULL (chosen from the training size).
use_fnn	Use **FNN** for nearest-neighbour search when available. Exposed so the dense fallback can be tested.

Value

An object of class aoa: a list with

- aoa – newdata with a numeric DI column and a logical AOA column added. This is the object the computation ran on, which for a coordinate-using model is newdata after pointizing, CRS reconciliation and the addition of the ".x" and ".y" columns. A row whose predictors are not all finite gets NA in both columns.

- `threshold` – the DI cut-off used.
- `train_DI` – the training points' own DI values.
- `normalizer` – the mean pairwise training distance.
- `weights` – the weight vector actually applied, named by `predictor_vars`.
- `predictor_vars` – the predictors used, including `"..x"/"..y"` when the model uses coordinates and excluding `dropped_vars`.
- `dropped_vars` – predictors dropped for negligible variance.
- `n_train`, `n_new`, `n_inside`, `n_outside`, `n_na` – row counts; `n_train` and `n_new` count the rows that survived the finite-value filter.
- `params` – a record of the call: `folds_supplied`, `n_folds`, `threshold_supplied`, `normalizer_max_n`, `normalizer_n_used`, `normalizer_subsampled`, `weights_supplied` and `seed`.

Why a map alone is not enough

A fitted model will return a number for any location you hand it, including locations whose predictor values look nothing like anything it was trained on. Those predictions are extrapolations dressed as interpolations, and a cross-validation score says nothing about them, because the held-out folds were drawn from the same predictor distribution as the training data. The AOA marks where the score applies.

How it is computed

Predictors are centred and scaled using the training data's own means and standard deviations, then optionally weighted by variable importance. For a prediction point p, the DI is the distance to its nearest training point in that space, divided by the mean pairwise distance among training points. The same quantity is computed for the training data itself, using each point's nearest neighbour *among the training rows of the fold that holds it out* – everything outside its own fold for random and block folds, the smaller training set that buffered and NNDM folds actually leave (see the next section) – and the threshold is the largest training DI that is not an upper outlier. Prediction points at or below that threshold are inside the AOA.

The DI is invariant to the overall scale of `weights`: the numerator and the normaliser carry the same factor. Importance values can be passed as-is.

The fold scheme changes the answer, and should

With `folds = NULL` the training reference is each point's nearest neighbour anywhere in the training data, which for clustered data is very close, giving a small threshold and a conservative AOA. Passing the folds you actually validated with makes the reference distances larger and the AOA correspondingly wider. That is not a loophole – the AOA is defined relative to a performance estimate, and a spatially blocked estimate is a claim about predicting further away. Pass the same `make_folds()` result you passed to `cv_spatial`. Buffered and NNDM folds use the training set they actually left available, not merely "everything outside the fold".

Limitations

Predictors must be numeric; categorical variables are refused rather than silently dummy-coded. Predictors whose variance is negligible *relative to their own magnitude* (the test is `sd < sqrt(.Machine$double.eps)`

* max(abs(x)), so the same variable in metres and in gigametres is treated identically) are dropped, and a prediction point taking a different value there is a form of extrapolation this index cannot express. Without weights every predictor counts equally, which overstates dissimilarity along directions the model barely uses.

Models fitted with the coordinates as predictors

When model was fitted with include_coords = TRUE the model splits on location, so the dissimilarity index has to measure location too: the coordinates are added to both sides as the predictors "..x" and "..y" and are then centred, scaled and weighted like any other column. Without this a prediction point far outside the training extent but with ordinary covariate values reads as *inside* the area of applicability – exactly the extrapolation this index exists to catch.

This path needs geometry on both sides, so train_sf and newdata must both be sf objects; a data.frame is refused rather than quietly measured without location. Non-POINT newdata (grid polygons, say) is reduced to representative points first, as coerce_to_points would. If exactly one side carries a CRS the other is brought into it – reprojected when its coordinates look like longitude/latitude, stamped otherwise, with a warning either way – because degrees fed into a metre-space index silently understate the distances.

References

Meyer, H. and Pebesma, E. (2021). Predicting into unknown space? Estimating the area of applicability of spatial prediction models. *Methods in Ecology and Evolution* 12(9), 1620–1633. doi:10.1111/2041210X.13650

See Also

predict_surface to build the grid, make_folds for the fold scheme.

Other cross-validation: cv_bayes(), cv_gwr(), cv_rf(), cv_spatial(), estimate_sac_range(), gwr_model_selection(), make_folds(), select_features_forward()

Examples

```
library(sf)
set.seed(1)
n <- 120
train <- st_as_sf(
  data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000),
             a = rnorm(n), b = rnorm(n)),
  coords = c("x", "y"), crs = 32632
)
train$z <- 2 * train$a - train$b + rnorm(n, 0, 0.3)

# Prediction points, some of them well outside the training predictor range
newpts <- st_as_sf(
  data.frame(x = runif(50, 0, 1000), y = runif(50, 0, 1000),
             a = c(rnorm(40), rnorm(10, 8)), b = rnorm(50)),
  coords = c("x", "y"), crs = 32632
)
```

```
res <- area_of_applicability(newpts, train_sf = train,
                             predictor_vars = c("a", "b"))
res
table(res$aoa$AOA)
```

`assign_features_to_polygons` *Assign features to polygons and attach a polygon ID*

Description

Joins an sf layer of input features to a polygon layer via spatial join.

Usage

```
assign_features_to_polygons(
  features_sf,
  polygons_sf,
  polygon_id_col = "poly_id",
  keep_unassigned = FALSE,
  predicate = sf::st_intersects,
  largest = TRUE,
  tie_break = c("smallest_area", "first")
)
```

Arguments

`features_sf`	An sf object containing features to assign.
`polygons_sf`	An sf or sfc polygonal layer.
`polygon_id_col`	Name of the polygon identifier column. Default "poly_id".
`keep_unassigned`	Logical; retain features that fall inside no polygon, carrying `NA` in the ID column. Default FALSE, which drops them.
`predicate`	Binary spatial predicate function. Default sf::st_intersects.
`largest`	Logical; when `features_sf` is itself polygonal, keep the polygon with the largest overlap. Default TRUE. Ignored for point and line features, and silently dropped if the `predicate` does not support it (`sf::st_intersects` does).
`tie_break`	Strategy for resolving features that match multiple polygons: `"smallest_area"` (default) keeps the polygon with the smallest area, `"first"` keeps the first match (original order-dependent behavior).

Details

This is the second step of the package's pipeline: it labels every observation with the cell it falls in, which is what `summarize_by_cell()` then aggregates over. Reach for it directly (rather than for `sf::st_join()`) when the join has to be *unambiguous* — it resolves features matching several polygons by an explicit `tie_break` rule instead of silently duplicating rows, so the assigned layer keeps one row per input feature and cell-level counts mean what they say.

Value

An sf object with `polygon_id_col` attached, one row per input feature (fewer if `keep_unassigned = FALSE` dropped unmatched ones), in the CRS `features_sf` arrived in. Any column of `features_sf` whose name would collide with the polygon ID column is dropped before the spatial join (with a warning), so re-assigning an already-assigned layer replaces the old IDs rather than failing. If *no* feature falls inside any polygon the result is empty (or all-`NA` with `keep_unassigned = TRUE`) and a warning is raised, since the usual cause is two layers in different places — a CRS that could only be stamped, not reprojected.

See Also

`build_tessellation()` to build the polygon layer; `summarize_by_cell()` for the aggregation step that consumes the result.

Other aggregation: `determine_optimal_levels()`, `summarize_by_cell()`

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = runif(20, 0, 100), y = runif(20, 0, 100), val = rnorm(20)),
  coords = c("x", "y"), crs = 32632
)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
grid <- create_grid_polygons(bnd, target_cells = 9, type = "square")
assigned <- assign_features_to_polygons(pts, grid)
table(assigned$poly_id)
```

`build_tessellation` *Build a tessellation (Voronoi, Delaunay triangles, hex grid, or square grid)*

Description

The single entry point for turning a point pattern into analysis regions, and the first step of the package's pipeline. It wraps the four tessellation methods behind one interface that handles CRS projection, clipping and stable cell identifiers consistently, and returns the cell layer together with the point-to-cell index that `assign_features_to_polygons()` and `summarize_by_cell()` consume. Use it rather than the individual constructors whenever you might want to compare methods: the return shape does not change with `method`, so swapping `"voronoi"` for `"hex"` costs one argument.

Usage

```
build_tessellation(
  points_sf,
  boundary = NULL,
  method = c("voronoi", "triangles", "hex", "square"),
  approx_n_cells = NULL,
  cellsize = NULL,
  expand = 0,
  clip = TRUE,
  keep_duplicates = FALSE,
  crs = NULL,
  quiet = FALSE
)
```

Arguments

points_sf	An sf object with POINT/MULTIPOINT geometry.
boundary	Polygonal sf/sfc study area. **Required** for method = "hex" and method = "square", which have no extent of their own to lay a grid over and error without it; supply the study-area polygon, or build one from the points with clip_target_for(). **Optional** for method = "voronoi" and method = "triangles", which derive their extent from the points themselves and use boundary only to clip the result when clip = TRUE.
method	One of "voronoi", "triangles", "hex", "square".
approx_n_cells	Approximate number of cells (grid methods). For hex grids the target is adjusted for packing density; the actual count after clipping to an irregular boundary may differ noticeably.
cellsize	Numeric cell size (grid methods).
expand	Buffer distance for the Voronoi envelope. Applied by method = "voronoi" only; the "hex", "square" and "triangles" methods ignore it (the value you passed is still echoed back in params$expand).
clip	Logical; clip to boundary.
keep_duplicates	Logical; keep duplicate points.
crs	Optional target CRS.
quiet	Logical; suppress this function's progress message()s. It does not silence R warnings, nor the package's console log echo (see spatialkit_quiet for that). Default FALSE.

Details

Which method to reach for. "voronoi" gives one cell per point, so resolution follows sampling density – the choice when the observations themselves define the regions. "hex" and "square" give equal-area cells on a fixed grid, so cell size is a decision you make rather than one the data makes for you; hexagons avoid the axis-aligned artefacts of squares and have uniform neighbour distances. "triangles" returns the Delaunay triangulation, useful for interpolation and adjacency

work rather than as an aggregation unit. `determine_optimal_levels()` will suggest a cell count from the spatial structure of the data.

Value

A list with components:

`cells` An sf polygon layer, one row per cell. It always carries a `cell_id` column; the `"hex"` and `"square"` methods additionally carry `poly_id`, which holds the same values.

`index` Integer vector of `cell_id` values, one per row of `points_sf`, and `NA` for a point that falls inside no cell — one outside the study area, in other words. Only a point within a thousandth of the median cell width of a cell is snapped to it, which covers points sitting exactly on a shared edge without quietly dragging genuinely-outside points in. A summary built from `index` therefore counts only the points the tessellation actually covers.

`boundary` The boundary used (possibly derived and/or reprojected).

`method` The method actually used.

`params` The parameters the tessellation was built with.

See Also

Other tessellation: `create_grid_polygons()`, `create_voronoi_polygons()`, `get_voronoi_seeds()`, `plot_tessellation_map()`

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = runif(20, 0, 100), y = runif(20, 0, 100)),
  coords = c("x", "y"), crs = 32632
)
tess <- build_tessellation(pts, method = "voronoi", quiet = TRUE)
tess$cells
```

---

clear_fitted_cache *Clear cached fitted values for a Bayesian spatial model*

Description

Removes the lazily-cached `fitted()` result so that the next call recomputes from the posterior. This is only necessary if the underlying `brmsfit` engine has been manually mutated after fitting – a change to `data_sf` invalidates the entry on its own, because the cached value carries a digest of the data it was computed from (see `fitted.bayesian_fit`). Normal usage never requires it.

Usage

```
clear_fitted_cache(object)
```

Arguments

object	A bayesian_fit object.

Details

The cache environment is shared by every copy of a fit, so clearing it through one copy clears it for all of them. That is harmless: the others recompute.

Value

object, invisibly (called for side effect).

Examples

```
# Only a bayesian_fit carries the cache; on any other fit this is a no-op.
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = runif(60, 0, 1000), y = runif(60, 0, 1000), a = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  pts$z <- 2 * pts$a + rnorm(60, 0, 0.3)
  fit <- fit_rf_model(pts, "z", "a", num_trees = 50, seed = 1)
  clear_fitted_cache(fit)
}
```

---

clear_grid_cache *Clear the in-session grid cache*

Description

Removes all memoized grid results from the internal cache environment.

Usage

```
clear_grid_cache(cache_env = .gmt_cache)
```

Arguments

cache_env	Environment to clear. Default .gmt_cache.

Value

Invisibly, the number of entries removed.

Examples

```
library(sf)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
g1 <- create_grid_polygons_cached(bnd, target_cells = 9)
g2 <- create_grid_polygons_cached(bnd, target_cells = 9)   # cache hit
clear_grid_cache()                                          # entries removed
```

`clip_target_for` *Build a polygonal clip target from points and/or a boundary*

Description

Resolves the single polygon that every tessellation method clips against. With a `boundary` it is that boundary (optionally buffered by `expand`); without one it is the convex hull of `points_sf`, again optionally buffered. Reach for it when you want to see or reuse the exact clip target `build_tessellation()` will apply — for instance to check that a study-area polygon actually contains the observations before tessellating, or to pass the same envelope to `create_voronoi_polygons()` and `create_grid_polygons()` so that two tessellations of one dataset cover identical ground.

Usage

```
clip_target_for(points_sf, boundary = NULL, expand = 0, quiet = FALSE)
```

Arguments

`points_sf`	An sf object with POINT/MULTIPOINT geometry.
`boundary`	Optional polygonal sf object.
`expand`	Numeric expansion distance or fraction (0–1 = fraction of extent). Absolute values are expressed in the units of the CRS the clip target is built in. Because `sf::st_buffer()` interprets `dist` as **metres** for lon/lat data while the fraction-of-extent form is derived from a bounding box measured in **degrees**, lon/lat input is projected to a local projected CRS first (see `@return`), so both forms agree.
`quiet`	Logical; suppress this function's progress `message()`s. It does not silence R warnings, nor the package's console log echo (see `spatialkit_quiet` for that). Default `FALSE`.

Value

An sf polygon layer representing the clip target. For lon/lat input the layer is returned in the automatically selected local projected CRS, not the input CRS; a message reports this unless `quiet = TRUE`.

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = runif(30, 0, 100), y = runif(30, 0, 100)),
  coords = c("x", "y"), crs = 32632
)
# No boundary: the convex hull, expanded by 10% of the extent
hull <- clip_target_for(pts, expand = 0.1, quiet = TRUE)
st_area(hull)
```

---

coef.bayesian_fit *Extract Bayesian model fixed-effect summaries*

Description

Returns the posterior summary of the global (non-spatial) regression terms: estimate, error and credible interval per predictor, as `brms::fixef()` reports them. Reach for it to read the average effect of a predictor with its uncertainty attached – the Bayesian counterpart to a coefficient table – remembering that the Gaussian-process term has already absorbed the spatially structured part of the signal, so these are effects net of location.

Usage

```
## S3 method for class 'bayesian_fit'
coef(object, ...)
```

Arguments

`object`	A `bayesian_fit` object.
`...`	Ignored.

Value

A matrix of fixed-effect posterior summaries, as returned by `brms::fixef()`. Never `NULL`: a missing 'brms' or a failing `fixef()` call errors, following the `coef()` contract described in `new_spatial_fit`.

`coef.gwr_fit` *Extract GWR local coefficients*

Description

Returns the whole surface of coefficients – one row per observation, one column per term – rather than the single global vector `coef()` returns for an `lm`. That table is the point of fitting a GWR at all: inspect the spread of a predictor's column to see where, and by how much, its relationship with the response changes across the study area, and join it back to `object$data_sf` to map it. Use `plot.spatial_fit()` for a quick look at that map.

Usage

```
## S3 method for class 'gwr_fit'
coef(object, ...)
```

Arguments

`object`	A `gwr_fit` object.
`...`	Ignored.

Value

A data.frame of local coefficient estimates: one row per observation, one column per model term. Never `NULL`: when the engine carries no `SDF` component this errors, following the `coef()` contract described in `new_spatial_fit`.

What is and is not returned

Only the model terms – the intercept and one column per predictor. GWmodel's `SDF` data slot carries a good deal more alongside them (standard errors, t-values, the observed response, the fitted values, the residuals, `Local_R2`): 15 columns for a two-predictor fit, of which 3 are coefficients. Returning the whole slot would have made `coef(fit)$Local_R2` and `coef(fit)$a_SE` read like coefficients and `ncol(coef(fit))` a meaningless number. Reach for `object$engine$SDF` when you want the rest; it is the unmodified GWmodel object.

If the model terms cannot be located in the `SDF` – a GWmodel that names its coefficient columns differently – the whole slot is returned with a warning saying so, rather than an error or a silently short table.

coef.rf_fit *Coefficients are undefined for a random forest*

Description

Consistent with `coef.gwr_fit()` and `coef.bayesian_fit()`, which also error rather than returning `NULL` when they cannot supply coefficients – see the `coef()` contract in `new_spatial_fit`.

Usage

```
## S3 method for class 'rf_fit'
coef(object, ...)
```

Arguments

`object`	An `rf_fit`.
`...`	Ignored.

Value

Never returns; always signals an error.

coerce_to_points *Coerce arbitrary geometries to representative points*

Description

Converts the geometry column of an sf object to POINTs using one of several strategies.

Usage

```
coerce_to_points(
  x,
  mode = c("auto", "centroid", "point_on_surface", "surface", "line_midpoint",
    "bbox_center"),
  tmp_project = TRUE
)
```

Arguments

`x`	An sf object.
`mode`	One of "auto", "centroid", "point_on_surface", "surface", "line_midpoint", "bbox_center".
`tmp_project`	Logical; temporarily project for line-based midpoints. When `x` has no CRS and its coordinates fall inside the lon/lat envelope, that temporary projection interprets them as EPSG:4326 (with a warning) and the midpoints returned are geodesic ones brought back to the input's numbers, not planar midpoints. Set the CRS, or pass `tmp_project = FALSE`, for planar data.

Details

LINESTRING midpoints are sampled with `sf::st_line_sample()`, which yields no point for an EMPTY LINESTRING. Rather than silently misaligning the result (or letting sf crash), such input raises an error; drop empty geometries first with `x <- x[!sf::st_is_empty(x), ]`.

Value

An sf object with geometry coerced to POINTs.

Examples

```
library(sf)
poly <- st_sf(
  id = 1,
  geometry = st_sfc(st_polygon(list(rbind(
    c(0, 0), c(2, 0), c(2, 2), c(0, 2), c(0, 0)
  ))), crs = 32632)
)
coerce_to_points(poly, "auto")  # interior representative point
```

`compare_models` *Side-by-side comparison of fitted spatial models*

Description

Takes a named list of already-fit `spatial_fit` objects and produces a tidy comparison table including in-sample metrics and model-specific information criteria (AICc, LOOIC).

Usage

```
compare_models(fits, newdata = NULL, ...)
```

Arguments

`fits`	A named list of `spatial_fit` objects. Names must be unique; see `evaluate_insample`.
`newdata`	Optional sf for out-of-sample evaluation.
`...`	Extra arguments passed to predict().

Value

A data.frame comparing all models. Alongside the metrics it carries `resid_morans_I`, `resid_morans_z`, `resid_morans_p` and `resid_morans_null` — the last naming which null `residual_morans_i` scored each model against, since that choice is per-fit and governs how much the p-value is worth. The significant-autocorrelation warning below is driven by that p-value, so read its caveats in `?residual_morans_i` before treating silence as evidence of no residual structure.

Percentage errors on responses with zeros

MAPE divides by the observed value and SMAPE by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return Inf or NaN, both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. **The returned value does not record how many rows that was**, and the n column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, MAPE is an average over the 58 non-zero rows reported as though it covered all 120. SMAPE fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

Other model evaluation: compare_models_cv(), evaluate_insample(), residual_morans_i()

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = runif(60, 0, 1000), y = runif(60, 0, 1000), a = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  pts$z <- 2 * pts$a + rnorm(60, 0, 0.3)
  fits <- list(RF_small = fit_rf_model(pts, "z", "a", num_trees = 50, seed = 1),
               RF_big   = fit_rf_model(pts, "z", "a", num_trees = 200, seed = 1))
  compare_models(fits)
}
```

---

compare_models_cv *Cross-validated comparison of spatial models*

Description

Fits and cross-validates one or more model types, returning a unified comparison table. Unlike compare_models(), this function does perform fitting (inside CV folds), because CV inherently requires repeated fitting.

Usage

```
compare_models_cv(
  data_sf,
  response_var,
  predictor_vars,
  models = c("GWR", "Bayesian"),
  k = 5,
  seed = 123,
  folds = NULL,
  boundary = NULL,
  pointize = "auto",
  gwr_args = list(),
  bayes_args = list(),
  rf_args = list(),
  summary = c("mean", "median"),
  quiet = FALSE
)
```

Arguments

data_sf	An sf object.
response_var	Response column name.
predictor_vars	Predictor column names.
models	Character vector: any subset of c("GWR", "Bayesian", "RF"), in any order. Each is cross-validated on the same folds. Names outside that set raise a warning and are dropped; if nothing recognised remains, this is an error rather than a silent fallback. A recognised model whose backend package is not installed is dropped with a message so the call still returns the models that could run — but if *none* of the requested backends is installed, nothing is left to compare and the call errors with "no viable models.". Guard with requireNamespace() when the model set is not known in advance.
k	Number of folds. Default 5.
seed	RNG seed. Default 123.
folds	Optional fold definitions: a make_folds() return value, or a bare list of list(train =, test =) pairs of ..row_id values. Train and test must be disjoint — a fold that trains on its own test rows is not a cross-validation split and is refused with an error — and IDs naming no row in the prepared data are dropped with a logged count (expected when rows were removed for missing values; a sign the folds came from other data when they were not).
boundary	Optional polygon sf/sfc.
pointize	Geometry coercion strategy.
gwr_args	Extra arguments for cv_gwr. Only names that are formal arguments of cv_gwr() are forwarded — it has no ... — so entries meant for fit_gwr_model() alone (e.g. longlat) cannot be passed this way. Anything dropped is named in a warning; call cv_gwr() directly if you need it.

bayes_args	Extra arguments for fit_bayesian_spatial_model(). Forwarded whole as cv_bayes(fit_args =), so an unrecognised name is an error from fit_bayesian_spatial_model() rather than a silent drop. compute_loo, boundary and pointize are overridden by the CV internals.
rf_args	Extra arguments for cv_rf, which passes anything it does not recognise on to fit_rf_model and thence to ranger::ranger().
summary	"mean" or "median" for Bayesian predictions.
quiet	Logical; suppress this function's progress message()s. It does not silence R warnings, nor the package's console log echo (see spatialkit_quiet for that). Default FALSE.

Value

A list with overall, by_fold, and per-model cv_results (gwr_cv, bayes_cv, rf_cv for the models that ran). Only the models that actually ran appear, so check which names are present rather than assuming one entry per requested model: a backend whose package is missing is dropped with a message. When **no** requested backend is available there is nothing to return and the function errors with "no viable models." instead of returning an empty comparison.

Percentage errors on responses with zeros

MAPE divides by the observed value and SMAPE by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return Inf or NaN, both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. **The returned value does not record how many rows that was**, and the n column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, MAPE is an average over the 58 non-zero rows reported as though it covered all 120. SMAPE fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

Other model evaluation: compare_models(), evaluate_insample(), residual_morans_i()

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
```

```
  )
  dat$price <- 10 + 0.01 * st_coordinates(dat)[, 1] + 2 * dat$elev + rnorm(n)
  cmp <- compare_models_cv(dat, "price", "elev", models = "RF", k = 3,
                           rf_args = list(num_trees = 100))
  cmp$overall
}
```

`create_grid_polygons` *Create square or hexagonal grid polygons over a boundary*

Description

Lays a regular grid of equal-area cells over `boundary` and clips it to that boundary. Reach for this rather than `create_voronoi_polygons()` when cell size should be a decision you make — because you need per-cell rates comparable across the map, or a resolution that stays fixed as the sample grows — instead of one dictated by where the observations happen to be. Hexagons (`type = "hex"`) avoid the axis-aligned artefacts of squares and give every cell the same distance to all six neighbours, which matters for anything that reads neighbourhoods.

Usage

```
create_grid_polygons(
  boundary,
  target_cells = NULL,
  type = c("square", "hex"),
  cellsize = NULL,
  n = NULL,
  clip = TRUE,
  crs = NULL,
  quiet = FALSE,
  max_cells = 1e+06
)
```

Arguments

`boundary`	Polygonal sf or sfc object.
`target_cells`	Optional approximate desired number of cells. The cell *size* is derived from it as `sqrt(area / target_cells)`, so square grids get genuinely square cells; for hex grids the count is adjusted for hexagonal packing density. The word "approximate" is load bearing: a grid of square cells over an elongated bounding box needs more of them than a grid of rectangles would (a 1000 x 1 strip at `target_cells = 9` yields cells of side 10.5 and about 95 of them), and clipping to an irregular boundary moves the count again. Pass `cellsize` when the count matters more than the shape.
`type`	Grid type: `"square"` (the default) or `"hex"`.

cellsize	Optional numeric cell size (length 1 or 2), in the units of the working CRS. Takes precedence over n: if both are supplied, cellsize is used, n is ignored and a warning is logged. Supply exactly one of target_cells, cellsize and n. For type = "hex" a hexagon is defined by a single edge-to-edge distance, so only cellsize[1] is used and a differing cellsize[2] is ignored with a logged warning.
n	Optional grid resolution (integer, length 1 or 2) giving the number of columns and rows to divide the boundary's bounding box into; the cell size is derived from it. sf::st_make_grid() derives hexagon placement from cellsize alone, so for type = "hex" n does not set the number of cells – but it does change the grid, because the cellsize derived from it is what the hexagons are built with. Ignored (with a logged warning) when cellsize is also supplied — passing both would otherwise truncate the grid to n[1] x n[2] cells anchored at the bounding-box corner, covering only part of the boundary.
clip	Logical; clip grid to boundary.
crs	Optional target CRS. When NULL (default) the boundary is projected with ensure_projected(), which changes the CRS of the returned grid; a message reports this unless quiet = TRUE.
quiet	Logical; suppress this function's progress message()s. It does not silence R warnings, nor the package's console log echo (see spatialkit_quiet for that). Default FALSE.
max_cells	Upper bound on the number of cells the grid may have, estimated from the boundary's bounding box before anything is built. Default 1e6. A cellsize in the wrong units – metres on a boundary in kilometres, say – asks for a grid that cannot be built, and this refuses it with a message rather than exhausting memory. Set to Inf to disable.

Details

Size the grid with exactly one of target_cells (roughly how many cells you want, the package derives the rest), cellsize (a fixed edge length in CRS units) or n (a fixed number of columns and rows). See @param cellsize for what happens when more than one is given.

Value

An sf polygon layer with poly_id column.

See Also

Other tessellation: build_tessellation(), create_voronoi_polygons(), get_voronoi_seeds(), plot_tessellation_map()

Examples

```
library(sf)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
```

```
grid_sq  <- create_grid_polygons(bnd, target_cells = 100, type = "square")
grid_hex <- create_grid_polygons(bnd, target_cells = 100, type = "hex")
nrow(grid_sq)
# Hex counts run above target because clipping keeps every hexagon that
# merely overhangs the boundary; the inflation is proportionally larger
# at small target_cells.
nrow(grid_hex)
```

`create_grid_polygons_cached` *Create and cache grid polygons over a boundary*

Description

Builds a grid via `create_grid_polygons()` and memoizes the result so repeated calls with the same inputs return instantly.

Usage

```
create_grid_polygons_cached(
  boundary,
  target_cells,
  type = c("square", "hex"),
  ...,
  cache_env = .gmt_cache,
  max_entries = 50L
)
```

Arguments

`boundary`	An sf or sfc polygonal object.
`target_cells`	Approximate desired number of cells.
`type`	Grid type: `"square"` (the default) or `"hex"`, matching `create_grid_polygons()`.
`...`	Additional arguments forwarded to create_grid_polygons().
`cache_env`	Environment for memoized grids. Default .gmt_cache.
`max_entries`	Maximum number of grids the cache holds. Default 50. Once full, adding a grid evicts the one added earliest, so a loop over many boundaries holds at most this many grids (about 2 MB per 2,500-cell grid) rather than every grid it ever built for the life of the session. `clear_grid_cache` empties it outright.

Value

An sf data frame with a stable poly_id column. Note that the rows are re-ordered and re-numbered by `ensure_stable_poly_id`, which `create_grid_polygons` does not do: the same cell therefore carries a different `poly_id` depending on which of the two builders produced it. Use one builder throughout an analysis; joining a summary keyed on IDs from one onto geometries from the other draws the values on the wrong polygons.

Examples

```
library(sf)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
g <- create_grid_polygons_cached(bnd, target_cells = 16, type = "hex")
nrow(g)
head(g$poly_id)   # stable IDs from ensure_stable_poly_id()
```

`create_voronoi_polygons` *Create Voronoi polygons from points with robust CRS and optional clipping*

Description

Assigns every location in the study area to its nearest input point, giving one cell per point. This is the tessellation to reach for when the observations themselves define the regions of interest — sampling sites, monitoring stations, service points — because cell size then adapts to sampling density instead of being imposed by a fixed grid: dense areas get small cells and sparse areas large ones. Prefer `create_grid_polygons()` instead when you need equal-area cells or a resolution independent of where the data happen to be.

Usage

```
create_voronoi_polygons(
  points_sf,
  boundary = NULL,
  expand = 0,
  clip = TRUE,
  keep_duplicates = FALSE,
  crs = NULL,
  quiet = FALSE
)
```

Arguments

`points_sf`	An sf object with POINT/MULTIPOINT geometries.
`boundary`	Optional polygonal sf object.
`expand`	Numeric; absolute buffer distance for the envelope.
`clip`	Logical; intersect cells with boundary.
`keep_duplicates`	Logical; keep coincident points for graph construction.
`crs`	Optional target CRS.
`quiet`	Logical; suppress this function's progress `message()`s. It does not silence R warnings, nor the package's console log echo (see `spatialkit_quiet` for that). Default `FALSE`.

Details

The heavy lifting is `sf::st_voronoi()`; what this adds is the surrounding bookkeeping — projecting lon/lat input, building and buffering an envelope so edge cells are bounded, clipping to `boundary`, restoring the point-to-cell correspondence that `st_voronoi()` scrambles, and stamping stable `cell_id` values.

Value

A list with `cells`, `index`, `boundary`, `method` and `params`. `index` holds one `cell_id` per row of `points_sf`, and `NA` for a point that falls outside every cell – outside the study area, in other words – so a summary built from it counts only the points the tessellation actually covers.

See Also

Other tessellation: `build_tessellation()`, `create_grid_polygons()`, `get_voronoi_seeds()`, `plot_tessellation_map()`

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = runif(15, 0, 100), y = runif(15, 0, 100)),
  coords = c("x", "y"), crs = 32632
)
res <- create_voronoi_polygons(pts, quiet = TRUE)
res$cells   # one polygon per unique point, with stable cell_id
res$index   # cell_id assignment for each input point
```

---

cv_bayes *K-fold cross-validation for the Bayesian spatial model*

Description

Refits the Gaussian-process model of `fit_bayesian_spatial_model()` on each training fold and scores it on the held-out fold. Beyond the point-prediction metrics the other CV wrappers report, this one scores the whole predictive *distribution*: `predictive_coverage` says what fraction of held-out observations fell inside the 50/80/95\ calibration together. That is the reason to reach for it – a Bayesian model is usually chosen for its uncertainty, and only held-out coverage shows whether those intervals are honest at locations the model has not seen.

Usage

```
cv_bayes(
  data_sf,
  response_var,
  predictor_vars,
  folds = NULL,
```

```
  k = 5,
  seed = 123,
  boundary = NULL,
  pointize = "auto",
  fit_args = list(),
  summary = c("mean", "median"),
  compute_pred_intervals = TRUE,
  coverage_levels = c(0.5, 0.8, 0.95),
  block_size = NULL,
  auto_range = FALSE,
  parallel = FALSE
)
```

Arguments

data_sf	An sf object.
response_var	Response column name.
predictor_vars	Predictor column names.
folds	Optional fold definitions, in any of three shapes: a make_folds() return value; a bare list of list(train =, test =) pairs of ..row_id values; or a vector of fold labels, one per row, which becomes leave-that-label-out splits. The label vector is how folds built by another package are used here – blockCV::cv_spatial() returns one as $folds_ids – since its $folds_list holds two *unnamed* vectors per fold and is refused by name. Train and test must be disjoint — a fold that trains on its own test rows is not a cross-validation split and is refused with an error — and IDs naming no row in the prepared data are dropped with a logged count (expected when rows were removed for missing values; a sign the folds came from other data when they were not).
k	Number of folds. Default 5.
seed	RNG seed. Default 123. It seeds fold construction **and**, through a per-fold draw, each fold's Stan sampler, so two seeds give different posteriors even on identical folds. A seed in fit_args overrides the per-fold draw with one fixed sampler seed.
boundary	Optional polygonal sf/sfc for CRS alignment.
pointize	Geometry coercion strategy.
fit_args	Named list of extra arguments for fit_bayesian_spatial_model(). A user-supplied gp_k is respected in every fold; when omitted, the GP rank is auto-selected per training fold. compute_loo, boundary, and pointize are always overridden by the CV internals.
summary	"mean" or "median" for posterior predictions.
compute_pred_intervals	Logical; compute predictive intervals.
coverage_levels	Numeric vector of coverage levels.
block_size	Optional minimum block edge length for spatial CV blocks (projected CRS units).

auto_range — Logical. If TRUE and folds is NULL, estimate the autocorrelation range and use it as the minimum block size. Default FALSE.

parallel — Logical or positive integer. If TRUE, auto-detect the number of cores and fit folds in parallel via parallel::mclapply() (macOS / Linux; falls back to sequential on Windows). If an integer > 1, use that many cores. Default FALSE (sequential). Bayesian folds with full MCMC runs are the primary beneficiary of this option.

Details

It is the most expensive wrapper in the package by a wide margin: every fold is a full MCMC run. Use few folds, and parallel = TRUE if you have the cores. For a cheap first pass on the same question, cross-validate a forest with cv_rf() and come back here once the predictor set has settled.

Value

A list with overall, fold_metrics, predictions, folds, n_folds_attempted, n_folds_succeeded, formula and predictive_coverage. The two fold counts make a run where every fold failed visible in the return value rather than only in a warning. predictions carries, beyond the columns its siblings share, yhat_sd: the posterior predictive standard deviation of each held-out row, from the same draws that give the coverage below (NA when compute_pred_intervals = FALSE or the draws failed for that fold). overall$Adj_R2 is always NA, as for every cv_*(): see cv_spatial. The predictive_coverage entries (one per coverage_levels value, plus mean_CRPS) are averages across folds **weighted by each fold's** n_pred, because the per-fold values in fold_metrics are themselves means over that fold's test rows; an unweighted average would not be the pooled quantity when fold sizes differ, which for spatially blocked folds they routinely do.

Percentage errors on responses with zeros

MAPE divides by the observed value and SMAPE by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return Inf or NaN, both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. **The returned value does not record how many rows that was**, and the n column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, MAPE is an average over the 58 non-zero rows reported as though it covered all 120. SMAPE fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

Other cross-validation: area_of_applicability(), cv_gwr(), cv_rf(), cv_spatial(), estimate_sac_range(), gwr_model_selection(), make_folds(), select_features_forward()

Examples

```
## Not run:
# Not run: fits with Stan, which needs a working C++ toolchain and takes
# minutes of MCMC -- both outside what an example may assume.
if (requireNamespace("brms", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 60
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * st_coordinates(dat)[, 1] + 2 * dat$elev + rnorm(n)
  cv <- cv_bayes(dat, "price", "elev", k = 2,
                 fit_args = list(chains = 2, iter = 500))
  cv$overall
  cv$predictive_coverage  # coverage at 50/80/95% plus mean CRPS
}

## End(Not run)
```

---

cv_gwr *K-fold cross-validation for GWR*

Description

Refits a geographically weighted regression from scratch on each training fold and scores it on the held-out fold, so the reported error is what the model achieves at locations it did not see. Reach for it whenever you need a defensible accuracy figure for a GWR: the in-sample R^2 that model_metrics() reports on a gwr_fit is close to meaningless, because a local regression with a small bandwidth can track the training points almost exactly. Bandwidth is re-selected per fold unless you fix it with bandwidth, which keeps the selection itself inside the cross-validation rather than tuning on the full data first.

Usage

```
cv_gwr(
  data_sf,
  response_var,
  predictor_vars,
  folds = NULL,
  k = 5,
  seed = 123,
  adaptive = TRUE,
  bandwidth = NULL,
  kernel = c("bisquare", "gaussian", "tricube", "boxcar", "exponential"),
  boundary = NULL,
  pointize = "auto",
```

```
  block_size = NULL,
  auto_range = FALSE,
  parallel = FALSE
)
```

Arguments

data_sf	An sf object.
response_var	Response column name.
predictor_vars	Predictor column names.
folds	Optional fold definitions, in any of three shapes: a `make_folds()` return value; a bare list of `list(train =, test =)` pairs of `..row_id` values; or a vector of fold labels, one per row, which becomes leave-that-label-out splits. The label vector is how folds built by another package are used here – `blockCV::cv_spatial()` returns one as `$folds_ids` – since its `$folds_list` holds two *unnamed* vectors per fold and is refused by name. Train and test must be disjoint — a fold that trains on its own test rows is not a cross-validation split and is refused with an error — and IDs naming no row in the prepared data are dropped with a logged count (expected when rows were removed for missing values; a sign the folds came from other data when they were not).
k	Number of folds. Default 5.
seed	RNG seed. Default 123.
adaptive	Logical; use adaptive bandwidth. Default TRUE.
bandwidth	Optional bandwidth, applied to every fold. For `adaptive = TRUE` an integer number of neighbours; for `adaptive = FALSE` a distance in the units of the **projected** CRS the folds are fitted in (geographic input is projected first, so a value in degrees is read as metres). `NULL` (default) selects a bandwidth per fold with `GWmodel::bw.gwr()`, which is reported in `fold_metrics$bandwidth`. See `fit_gwr_model`.
kernel	Kernel function type.
boundary	Optional polygonal sf/sfc for CRS alignment.
pointize	Geometry coercion strategy.
block_size	Optional minimum block edge length for spatial CV blocks (projected CRS units). Ensures blocks are at least as large as the spatial autocorrelation range.
auto_range	Logical. If `TRUE` and `folds` is `NULL`, estimate the autocorrelation range and use it as the minimum block size. Default `FALSE`.
parallel	Logical or positive integer. If `TRUE`, auto-detect the number of cores and fit folds in parallel via `parallel::mclapply()` (macOS / Linux; falls back to sequential on Windows). If an integer > 1, use that many cores. Default `FALSE` (sequential).

Details

Folds default to spatial blocks (`make_folds(method = "block_kfold")`), not random ones – with autocorrelated data a random split leaves a held-out point's neighbours in the training set and the score comes back flattering. Use `cv_bayes()` for the same treatment of a Bayesian GP model, `cv_rf()` for a forest, and `compare_models_cv()` to score several backends on one set of folds.

Value

A list with `overall`, `fold_metrics`, `predictions`, `folds`, `n_folds_attempted`, `n_folds_succeeded`, `formula` and `adaptive`. The two fold counts make a run where every fold failed visible in the return value rather than only in a warning, since `overall` is a well-formed all-`NA` row either way. `Adj_R2` is `NA` in both `overall` and `fold_metrics`: the pooled predictions have no single parameter count, and a GWR's effective parameter count is not its predictor count (see `cv_spatial`).

Percentage errors on responses with zeros

`MAPE` divides by the observed value and `SMAPE` by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return `Inf` or `NaN`, both are averaged over the rows whose denominator is non-zero, and are `NA` when no row qualifies. **The returned value does not record how many rows that was**, and the `n` column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, `MAPE` is an average over the 58 non-zero rows reported as though it covered all 120. `SMAPE` fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

`RMSE`, `MAE` and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, `cv_bayes()` additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

Other cross-validation: `area_of_applicability()`, `cv_bayes()`, `cv_rf()`, `cv_spatial()`, `estimate_sac_range()`, `gwr_model_selection()`, `make_folds()`, `select_features_forward()`

Examples

```
if (requireNamespace("GWmodel", quietly = TRUE) &&
    requireNamespace("sp", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 60
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * st_coordinates(dat)[, 1] + 2 * dat$elev + rnorm(n)
  cv <- cv_gwr(dat, "price", "elev", k = 3, bandwidth = 30)
  cv$overall
  cv$fold_metrics
}
```

cv_rf *Cross-validate a random forest with spatial folds*

Description

Thin wrapper over cv_spatial that refits a ranger forest on each training fold. Unlike the out-of-bag error, this holds out whole spatial blocks, so neighbours of a held-out point are not sitting in the training set.

Usage

```
cv_rf(
  data_sf,
  response_var,
  predictor_vars,
  folds = NULL,
  k = 5,
  seed = 123,
  parallel = FALSE,
  block_size = NULL,
  auto_range = FALSE,
  boundary = NULL,
  pointize = "auto",
  ...
)
```

Arguments

data_sf	An sf object.
response_var	Response column name.
predictor_vars	Predictor column names.
folds	Optional fold definitions, in any of three shapes: a make_folds() return value; a bare list of list(train =, test =) pairs of ..row_id values; or a vector of fold labels, one per row, which becomes leave-that-label-out splits. The label vector is how folds built by another package are used here – blockCV::cv_spatial() returns one as $folds_ids – since its $folds_list holds two *unnamed* vectors per fold and is refused by name. Train and test must be disjoint — a fold that trains on its own test rows is not a cross-validation split and is refused with an error — and IDs naming no row in the prepared data are dropped with a logged count.
k	Number of folds when folds is NULL. Default 5.
seed	RNG seed. Default 123. It seeds fold construction **and**, through a per-fold draw, each fold's forest — so two different seeds give different results even on identical folds. To grow every fold's forest from one fixed ranger seed instead, call cv_spatial with your own fit_fn wrapping fit_rf_model(seed =).

parallel
: Passed to cv_spatial. Default FALSE. Under forked workers each fold's forest runs single-threaded unless num_threads is passed explicitly, so parallel = 4 means four threads in total rather than four times the session's mc.cores.

block_size, auto_range, boundary
: Passed to cv_spatial.

pointize
: How non-POINT geometry is reduced to a point before fitting; passed to cv_spatial. Default "auto".

...
: Passed to fit_rf_model on every fold — num_trees, mtry, importance, include_coords and so on. data_sf, response_var, predictor_vars and .already_prepped are set by this function and must not be passed here (every fold would fail with "matched by multiple actual arguments"). A seed given here overrides the per-fold draw described above.

Value

The cv_spatial result.

Percentage errors on responses with zeros

MAPE divides by the observed value and SMAPE by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return Inf or NaN, both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. **The returned value does not record how many rows that was**, and the n column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, MAPE is an average over the 58 non-zero rows reported as though it covered all 120. SMAPE fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

Other cross-validation: area_of_applicability(), cv_bayes(), cv_gwr(), cv_spatial(), estimate_sac_range(), gwr_model_selection(), make_folds(), select_features_forward()

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 200
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), a = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$z <- 2 * dat$a + rnorm(n, 0, 0.3)
```

```
  cv_rf(dat, "z", "a", k = 4)$overall
}
```

`cv_spatial` *Model-agnostic spatial cross-validation*

Description

Run K-fold CV for any model that returns a `spatial_fit` object. This is the extensibility point: to plug in a new model type, supply a `fit_fn(train_sf)` that returns a `spatial_fit`.

Usage

```
cv_spatial(
  data_sf,
  response_var,
  predictor_vars,
  fit_fn,
  folds = NULL,
  k = 5,
  seed = 123,
  boundary = NULL,
  pointize = "auto",
  predict_args = list(),
  fold_info_fn = NULL,
  p = NULL,
  block_size = NULL,
  auto_range = FALSE,
  parallel = FALSE,
  .caller = "cv_spatial"
)
```

Arguments

`data_sf`	An sf object.
`response_var`	Response column name.
`predictor_vars`	Predictor column names.
`fit_fn`	A function of one argument, the training slice of `data_sf`, returning a `spatial_fit` built with `new_spatial_fit()`. It is called once per fold on the training rows only, so anything done inside it – scaling, tuning, an inner variable sweep – is already nested and leak-free. The `subclass` it stamps must have a `predict.<subclass>()` method registered, because that is how each fold is scored.
`folds`	Optional fold definitions, in any of three shapes: a `make_folds()` return value; a bare list of `list(train =, test =)` pairs of `..row_id` values; or a vector of fold labels, one per row, which becomes leave-that-label-out splits. The label vector is how folds built by another package are used here – `blockCV::cv_spatial()`

	returns one as `$folds_ids` – since its `$folds_list` holds two *unnamed* vectors per fold and is refused by name. Train and test must be disjoint — a fold that trains on its own test rows is not a cross-validation split and is refused with an error — and IDs naming no row in the prepared data are dropped with a logged count. Built via `block_kfold` when `NULL`.
`k`	Number of folds.
`seed`	RNG seed.
`boundary`	Optional boundary for fold construction.
`pointize`	Geometry coercion strategy.
`predict_args`	Extra arguments for predict().
`fold_info_fn`	Optional function for per-fold extras.
`p`	Number of predictors for Adj R² (NULL to skip). Only meaningful for models with a fixed global parameter count; pass NULL for models with spatially varying coefficients (e.g. GWR).
`block_size`	Optional minimum block edge length for spatial CV blocks (projected CRS units).
`auto_range`	Logical. If `TRUE` and `folds` is `NULL`, estimate the autocorrelation range and use it as the minimum block size. Default `FALSE`.
`parallel`	Logical or positive integer. If `TRUE`, auto-detect the number of cores and fit folds in parallel via `parallel::mclapply()` (macOS / Linux; falls back to sequential on Windows). If an integer > 1, use that many cores. Default `FALSE` (sequential).
`.caller`	Internal. The name the messages carry, so a wrapper such as `cv_rf` reports itself rather than `cv_spatial()`.

Value

A list with `overall`, `fold_metrics`, `predictions`, `folds`, and the two fold counts `n_folds_attempted` and `n_folds_succeeded`. The counts are reported deliberately: a `fit_fn` that fails on every fold otherwise looks like a successful run that happened to score `NA`, so compare them before trusting `overall`. The `fold` column of `fold_metrics` and `predictions` carries the fold's index in the `folds` object that was supplied, so it lines up with `make_folds()$assignment$fold` even when some folds were unusable and dropped. `overall$Adj_R2` is always `NA`: the pooled out-of-sample predictions come from `k` separately fitted models and have no single parameter count to adjust for. The per-fold `fold_metrics$Adj_R2` carries the adjusted value when `p` is supplied, and is `NA` otherwise.

Percentage errors on responses with zeros

`MAPE` divides by the observed value and `SMAPE` by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return `Inf` or `NaN`, both are averaged over the rows whose denominator is non-zero, and are `NA` when no row qualifies. **The returned value does not record how many rows that was**, and the `n` column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, `MAPE` is an average over the 58 non-zero rows

reported as though it covered all 120. SMAPE fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

new_spatial_fit() for the constructor a fit_fn must use; cv_gwr(), cv_bayes() and cv_rf() for the built-in backends, which are thin wrappers over this function.

Other cross-validation: area_of_applicability(), cv_bayes(), cv_gwr(), cv_rf(), estimate_sac_range(), gwr_model_selection(), make_folds(), select_features_forward()

Examples

```
library(sf)
set.seed(1)
n <- 80
site <- st_as_sf(
  data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
  coords = c("x", "y"), crs = 32632
)
site$price <- 10 + 0.01 * st_coordinates(site)[, 1] + 2 * site$elev + rnorm(n)

# 1. A fit_fn returning a spatial_fit of your own subclass.
lm_fit <- function(train_sf) {
  new_spatial_fit(
    subclass       = "lm_fit",
    engine         = lm(price ~ elev, st_drop_geometry(train_sf)),
    formula        = price ~ elev,
    response_var   = "price",
    predictor_vars = "elev",
    data_sf        = train_sf
  )
}

# 2. The predict() method cv_spatial() scores each fold with. Without it
#    every fold fails and `overall` comes back all-NA.
predict.lm_fit <- function(object, newdata = NULL, ...) {
  if (is.null(newdata)) newdata <- object$data_sf
  as.numeric(stats::predict(object$engine, st_drop_geometry(newdata)))
}
registerS3method("predict", "lm_fit", predict.lm_fit)

cv <- cv_spatial(site, "price", "elev", fit_fn = lm_fit, k = 3, seed = 1)
cv$overall
# Compare these before trusting the metrics above.
c(attempted = cv$n_folds_attempted, succeeded = cv$n_folds_succeeded)
```

`determine_optimal_levels` *Determine an optimal number of spatial levels via an elbow heuristic*

Description

Computes a WSS curve over k=1..K_max using k-means on projected feature coordinates and selects candidate k values around the elbow.

Usage

```
determine_optimal_levels(
  data_sf,
  max_levels = 12L,
  top_n = 3L,
  sample_n = 1500L,
  set_seed = 123L,
  response_var = NULL,
  predictor_vars = NULL,
  criterion = c("geometric", "morans_i", "combined")
)
```

Arguments

`data_sf`	An sf object.
`max_levels`	Integer upper bound on levels. Default 12.
`top_n`	Integer; how many candidates to return. Default 3. Under `criterion = "geometric"` the candidate set is the elbow and its two immediate neighbours, so at most 3 values are ever returned no matter how large `top_n` is; only the model-aware criteria can return more.
`sample_n`	Integer; subsample size for speed. Default 1500.
`set_seed`	Integer RNG seed. Default 123.
`response_var`	Optional response column name. When provided alongside `predictor_vars`, enables model-aware level selection via Moran's I on OLS residuals. Must be numeric or logical (logicals are read as 0/1); a factor or character response raises an error rather than being coerced, because the residuals of an OLS fit to arbitrary level codes carry no meaning to test for autocorrelation.
`predictor_vars`	Optional predictor column names. Must be numeric or logical (logicals are read as 0/1); factor/character columns raise an error.
`criterion`	One of `"geometric"` (default when no response given), `"morans_i"` (select the k whose residual Moran's I is least *significant*), or `"combined"` (rank-average of WSS elbow distance and that same quantity). Falls back to `"geometric"` if response/predictors are unavailable, and also when no candidate clears the nine-cell resolution floor described in **Details**.

Details

When `response_var` and `predictor_vars` are provided, the geometric WSS elbow is supplemented with Moran's I computed on OLS residuals at each candidate k. The Moran's I profile measures how much spatial autocorrelation in the response remains *unexplained* at a given tessellation resolution — a direct reflection of the spatial process being modeled, rather than mere geometric compactness of coordinates. The combined criterion selects the k that best balances geometric parsimony and residual spatial independence.

To keep memory use and runtime bounded for large `max_levels`, the initial k-means sweep records only within-cluster sum-of-squares (WSS) without retaining cluster assignments. Moran's I is then evaluated lazily: k-means is re-run only for a focused neighbourhood around the elbow (±4 by default, or `±top_n` if larger), so that only the most promising candidate k values incur the cost of the full Moran's I computation.

The model-aware criteria rank on the standardised deviate, not on |Moran's I|. Both $E[I]$ and $Var[I]$ depend on the number of cells, so $|I|$ falls as k grows whether or not the finer tessellation is capturing anything. Measured over 300 replicates of a response with *no* spatial structure, mean $|I|$ fell monotonically from 0.114 at `k = 10` to 0.050 at `k = 60` (-56%), which made an $|I|$ ranking prefer the largest candidate for arithmetic reasons alone. Candidates are therefore ordered by $|z| = |I - E[I]|/\mathrm{sd}(I)$ using the Cliff & Ord regression residual moments — exact here, because the cell-level residuals are OLS residuals by construction. Over the same runs z had mean ≈ 0, $\mathrm{sd} \approx 1$ and a two-sided 5\ 0.040–0.057 at every k. Both quantities are reported in the `"diagnostics"` attribute, as `moran_i` and `moran_z`.

Resolution floor on the model-aware criteria. Moran's I is computed on cell-level residuals with an 8-nearest-neighbour weight matrix, so it only carries information once there are more than nine cells. At nine or fewer, every cell is a neighbour of every other, the row-standardised weight matrix is complete, and Moran's I collapses to exactly $-1/(k-1)$ for *any* residual vector — a function of k alone. The criterion ranks on $|z|$, not on $|I|$, and at the floor the residual moments give $E[I] = I$ and $\mathrm{Var}[I] = 0$ identically (the algebra holds to 10^{-16}), so the standardised deviate is $0/0$: it carries no information about the tessellation, and whichever way rounding noise resolves it those candidates would rank first or last on nothing. They therefore return `NA` and are excluded from the model-aware ranking. When no candidate in the elbow neighbourhood clears the floor — which is the usual outcome for small `max_levels` — the whole call falls back to the geometric ranking and logs a warning; raise `max_levels` above roughly 10 if you want the model-aware criteria to contribute. Under `criterion = "combined"`, a candidate below the floor that sits alongside candidates above it is ranked last on the Moran's I axis while still competing on the geometric axis.

Value

An integer vector of candidate level counts, **best first**: under the geometric criterion the elbow, then its lower and upper neighbours; under the model-aware criteria the candidates in rank order. `k[1]` is therefore the top-ranked count on every path, and `top_n = 1` returns it alone. When `criterion != "geometric"`, an attribute `"diagnostics"` is attached with per-k Moran's I values (`moran_i`) and their standardised deviates (`moran_z`) — except when the model-aware path itself falls back to the geometric result (no viable k in the elbow neighbourhood, or Moran's I could not be computed for any candidate), in which case no diagnostics are available and the attribute is absent. Both fallbacks are logged as warnings.

See Also

build_tessellation(), which takes the chosen level count as approx_n_cells; assign_features_to_polygons() and summarize_by_cell() for the steps that follow.

Other aggregation: assign_features_to_polygons(), summarize_by_cell()

Examples

```
library(sf)
set.seed(1)
# Two clearly separated clusters: the elbow should sit near k = 2
pts <- st_as_sf(
  data.frame(x = c(runif(25, 0, 10), runif(25, 90, 100)),
             y = c(runif(25, 0, 10), runif(25, 90, 100))),
  coords = c("x", "y"), crs = 32632
)
determine_optimal_levels(pts, max_levels = 6)   # 2 1 3: the elbow first
```

---

ensure_projected *Ensure an object has a projected CRS (with sensible defaults)*

Description

Coerces spatial objects to a projected coordinate reference system suitable for distance/area calculations.

Usage

```
ensure_projected(x, target_crs = NULL)
```

Arguments

x	An sf or sfc object (other objects returned unchanged).
target_crs	Optional target CRS (sf object, integer EPSG, or crs). Must resolve to a usable CRS via sf::st_crs(); an unusable value (one that resolves to NA_crs_) raises an error rather than silently leaving x unprojected.

Details

An object that already has a projected CRS is returned untouched. Only geographic (lon/lat) input is transformed, and the CRS chosen depends on the extent of the data — it is **not** always UTM:

Local extents The UTM zone containing the data's centre (EPSG:326xx north of the equator, EPSG:327xx south). Distances and areas are close to true over a few degrees of longitude, which is the case this package is usually in.

Wide extents Once the data reach well beyond the roughly 3 degrees a UTM zone is designed for, a single zone can distort distances by several percent — and that error propagates straight into variogram ranges, block sizes, GWR bandwidths and GP length-scales. Which projection is actually best is then **measured, not assumed**: the zone, a Lambert azimuthal equal-area centred on the data and (where its standard parallels do not degenerate) an Albers conic are each scored by projecting representative points of the data — a non-POINT layer is reduced to points first — and comparing planar with geodesic pairwise distances, and the one that distorts least is used. The choice, both error figures and this argument are **logged** (see the logging note under `spatialkit_quiet()`); they are not R warnings, so `tryCatch(warning = )` does not see them.

Antimeridian Data straddling ±180° have a bounding box wider than a hemisphere. The wrap is detected from the coordinates (one very large gap in the sorted longitudes) and an equal-area projection centred on the true extent is used. Only genuinely global coverage falls back to EPSG:3857.

Missing CRS With no `target_crs`, a bounding box that looks like lon/lat means EPSG:4326 is assumed (a real warning) and the rules above then apply; coordinates the heuristic declines are left exactly as they are. With `target_crs` supplied there is no source CRS to reproject from, so the same heuristic decides between two outcomes: lon/lat-looking coordinates are read as EPSG:4326 and reprojected to the target (a real warning), and anything else has the target **stamped on without reprojection** — a relabel, logged only, so verify the coordinates really are in that CRS. Set the CRS explicitly to suppress either.

`target_crs` overrides all of this: pass it whenever you need a specific, reproducible projection — comparing runs, matching an existing layer, or fixing the units that `make_folds()`'s `block_size` will be interpreted in.

Value

x, potentially with a new projected CRS. CRS-less input additionally carries `attr(x, "crs_assumed")`: `"EPSG:4326"` when the lon/lat heuristic fired, `"none"` when it declined. That attribute is also read on the way IN — an object already carrying `"none"` is returned untouched, with the heuristic skipped, which is how a `predict()` method replays a fit's negative decision so that a subset of the training rows is not judged differently from the whole.

Examples

```
library(sf)
pts_ll <- st_as_sf(
  data.frame(lon = c(9.1, 9.2), lat = c(48.7, 48.8)),
  coords = c("lon", "lat"), crs = 4326
)
# A local extent gets the containing UTM zone.
st_crs(ensure_projected(pts_ll))$epsg  # 32632

# A continental extent is scored against the zone and may get an equal-area
# projection instead; the choice and both error figures are LOGGED, not
# warned -- see Details and ?spatialkit_quiet.
wide <- st_as_sf(
  data.frame(lon = c(-120, -70), lat = c(30, 48)),
  coords = c("lon", "lat"), crs = 4326
```

```
)
st_crs(ensure_projected(wide))$proj4string

# target_crs overrides the choice entirely.
st_crs(ensure_projected(pts_ll, target_crs = 3035))$epsg  # 3035
```

`ensure_stable_poly_id` *Create deterministic, stable polygon IDs based on spatial sort keys*

Description

Ensures that a polygon layer has a reproducible, deterministic identifier column by sorting features using representative point coordinates (and secondary tie-breakers) and then assigning sequential IDs.

Usage

```
ensure_stable_poly_id(
  polygons_sf,
  id_col = "poly_id",
  method = c("centroid", "surface_point", "bbox_center"),
  make_valid = TRUE,
  transform_for_sort = 4326
)
```

Arguments

`polygons_sf`	An sf or sfc object containing polygonal features.
`id_col`	Character scalar; name of the identifier column.
`method`	One of "centroid", "surface_point", "bbox_center".
`make_valid`	Logical; apply st_make_valid() first. Default TRUE.
`transform_for_sort`	CRS used only for computing sort-key coordinates. Default 4326. This is the whole mechanism by which the IDs are stable — sorting in one common CRS is what makes the same layer get the same IDs whichever projection it arrives in — so if the transform fails the function says so rather than quietly sorting in the input's own CRS. The sort key is rounded to 7 decimal degrees (about 1 cm) before ordering, so the floating-point noise of a round trip through a different projection cannot reverse two neighbouring cells. Set to NULL to sort in the input CRS, which gives IDs that are reproducible but not comparable across projections.

Value

An sf polygon layer re-ordered with sequential IDs in id_col. Non-polygonal rows are **dropped** (with a warning), so the result can have fewer rows than the input; if no polygonal rows remain, an error is raised.

Examples

```
library(sf)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
g <- create_grid_polygons(bnd, target_cells = 9)
# Reverse the rows: the IDs come back in the same spatial order regardless
ids_fwd <- ensure_stable_poly_id(g)$poly_id
ids_rev <- ensure_stable_poly_id(g[nrow(g):1, ])$poly_id
identical(sort(ids_fwd), sort(ids_rev))
```

---

estimate_sac_range *Estimate the spatial autocorrelation range from data*

Description

Fits exponential (or spherical) variogram models and returns the *effective range*: for the exponential model, three times the fitted range parameter, which is where the semivariance reaches ~95 \ sill; for the spherical model – fitted only when the exponential fit is singular – the fitted range itself, which is where the spherical semivariance reaches its sill exactly. Both are the distance beyond which two observations are (near) uncorrelated, which is what a block or a buffer has to exceed.

Usage

```
estimate_sac_range(
  points_sf,
  response_var,
  predictor_vars = NULL,
  n_max = 5000L,
  cutoff = 0.5,
  range_frac = 1,
  seed = 123L
)
```

Arguments

points_sf	An sf object with point geometries (will be projected automatically if in geographic CRS). Non-POINT geometry is reduced to representative points; any Z or M dimension is dropped, because `gstat::variogram()` uses every coordinate dimension and an XYZ layer would otherwise return a range in 3-D while every consumer of it works in 2-D map distance; and rows with empty or non-finite coordinates are dropped with a logged count.
response_var	Character(1) name of the response column.
predictor_vars	Optional character vector. When supplied, an OLS residual variogram is fitted instead of a raw-response variogram, which better reflects the autocorrelation that the spatial model must handle.

`n_max` Maximum number of points to subsample before fitting. Variogram estimation is O(n²) so this keeps runtime bounded.

`cutoff` Fraction of the maximum inter-point distance (the farthest pair, found on the convex hull – not the bounding-box diagonal, which depends on how the axes are oriented) to use as the variogram lag cutoff. Default 0.5.

`range_frac` Positive numeric. A fitted range exceeding `range_frac * cutoff * max_dist` – that is, beyond the longest lag the empirical variogram was actually fitted over – is treated as unidentified and `NA_real_` is returned. `gstat::fit.variogram()` yields a finite number even when the variogram never reaches a sill, and such a value is extrapolation past the observed lags rather than a long autocorrelation range. Passing it to `make_folds(auto_range = TRUE)` would collapse the block grid to a single block. Default 1.0; raise it to accept ranges extrapolated beyond the fitted lags.

`seed` RNG seed for the `n_max` subsample, restored afterwards so the caller's random stream is untouched. Default `123L`: the subsample is an internal approximation rather than part of the answer, and leaving it unseeded made the returned range differ between runs on identical input (19531, 19589, 19605 on three calls) and silently advanced the caller's RNG. Pass `NULL` for the old unseeded behaviour, or a different number to check how sensitive the estimate is to the subsample. Ignored when `nrow(points_sf) <= n_max`, where nothing is sampled.

Details

The estimate is the **omnidirectional** (all-pairs) fit. Directional variograms are fitted as well, at 0° (N–S), 45°, 90° (E–W) and 135° azimuths with a ±22.5° tolerance – four windows that tile all 180 distinct azimuths exactly once – and their ranges are returned in the `directional` attribute, with their largest-over-smallest ratio in `anisotropy`. They are a diagnostic, not the answer, for two reasons. Each direction sees about a quarter of the point pairs, and the maximum of four quarter-sample fits is biased upward: on simulated *isotropic* fields it came in about 40\ front of it (all four directions fitted, ratio above 1.5, maximum above 1.5× the all-pairs fit) kept it out – one isotropic field rotated in 10° steps "established" anisotropy in 14 of 18 orientations. And the windows are fixed to the coordinate axes, so any answer built from them changes when the layer is rotated, which a property of the field must not do. The all-pairs fit is the best-powered estimate available and is invariant to rotation.

Where a field is *known* to be anisotropic, blocks must be at least as large as the longest autocorrelation range to avoid leakage, and the conservative choice is to size them from `max(attr(range, "directional"))` explicitly. A ratio above 1.5 is logged so the case is not missed, with that advice. Only when the omnidirectional fit is itself unusable is the directional maximum returned in its place, and `anisotropy_used` is `TRUE` in that case alone.

A direction whose fit fails, does not converge, or reports a range beyond the longest fitted lag is excluded and recorded as `NA` in the `directional` attribute.

Every variogram model is fitted **with a nugget**. A nugget-free model forces the curve through the origin, and on any real measurement (which has one) **gstat**'s default N/h² weights buy that constraint by collapsing the range: with a 50% nugget the fitted range came back at about 0.45 of the truth, so `make_folds(auto_range = TRUE)` built blocks less than half the correlation length it reported.

A log warning is emitted when the directional maximum is used; where the all-pairs estimate is available it names both the ratio and that estimate. A log note is emitted instead when the directional

ranges vary but the spread is consistent with sampling noise.

The returned range is in the coordinate units of the (projected) data and can be passed directly to `make_folds(block_size = ...)` to ensure that CV blocks are at least as wide as the autocorrelation range.

Value

A single number, of class `sac_range` in the first two of the three shapes below and a bare `NA` in the third; all three behave as an ordinary number. The shapes carry different attributes:

Success A positive effective range in projected coordinate units, with the fit attached as attributes `directional` (the 0°, 45°, 90° and 135° ranges, named by azimuth), `anisotropy` (largest over smallest), `anisotropy_used` (logical: `TRUE` only when the all-pairs fit was unusable and the directional maximum stands in for it), `detrended` (logical: whether the variogram is of the OLS residuals on `predictor_vars` rather than the raw response – a missing predictor is an error, and a failed detrending fit warns and falls back to the raw response with this set to `FALSE`), `crs` (the projected CRS the variogram was fitted in – the unit of the range), `max_dist`, `cutoff_dist`, `variogram` (the empirical variogram) and `variogram_model` (the fitted `gstat` model), so the fit can be inspected rather than trusted.

Rejected range `NA_real_` when a range was fitted but exceeds `range_frac * cutoff * max_dist` and is therefore unidentified (see `range_frac`). It is classed `sac_range` as well, so it prints as a bare `NA` rather than dumping its attributes, and it carries `max_dist`, `cutoff_dist`, `variogram` and `variogram_model` — the evidence for the rejection — plus `rejected_range` (the value that was refused) and `rejected_reason`, plus `crs` — so the units the rejected number was in stay recoverable, which is what `plot(type = "variogram")` labels its axis from. It does **not** carry `directional` or `anisotropy`. The same shape, with `rejected_range = NA` and `variogram_model = NULL`, is returned when no variogram model could be fitted at all (both the exponential and the spherical fit singular, which is what a flat, nugget-only variogram produces); `rejected_reason` says so and the empirical variogram is still attached.

No fit A bare, attribute-less `NA_real_` when estimation could not be attempted at all: **gstat** missing, fewer than 30 finite values, a variable with no variance, or a degenerate extent. Without **gstat** nothing is fitted, so none of the attributes above exist either.

Attributes and the class do not affect `is.na()` or `is.finite()`, so every downstream guard treats all three the same way it always did.

See Also

Other cross-validation: `area_of_applicability()`, `cv_bayes()`, `cv_gwr()`, `cv_rf()`, `cv_spatial()`, `gwr_model_selection()`, `make_folds()`, `select_features_forward()`

Examples

```
if (requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  # A Gaussian random field with an exponential covariance: range
  # parameter 100, so the true effective range is 3 x 100 = 300, plus a
  # small nugget.
  set.seed(9)
```

```
  n <- 150
  xy <- data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000))
  D  <- as.matrix(dist(xy))
  xy$z <- as.numeric(t(chol(exp(-D / 100) + diag(0.1, n))) %*% rnorm(n))
  pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
  r <- estimate_sac_range(pts, response_var = "z")
  r                              # the effective range, in metres
  attr(r, "directional")         # the four directional ranges
  attr(r, "variogram_model")     # the fitted gstat model behind it

  # A field whose range the data cannot pin down: the variogram never
  # reaches a sill within the lags fitted, so the answer is NA with the
  # refused value attached rather than a long range asserted.
  xy$trend <- sin(xy$x / 400) + rnorm(n, sd = 0.2)
  r2 <- estimate_sac_range(st_as_sf(xy, coords = c("x", "y"), crs = 32632),
                           response_var = "trend")
  r2
  attr(r2, "rejected_range")
}
```

`evaluate_insample` *Compute in-sample (or out-of-sample) metrics for fitted spatial models*

Description

Accepts a single `spatial_fit` object or a named list of them. Does NOT refit — uses `fitted()` for in-sample and `predict()` for new data.

Usage

```
evaluate_insample(fits, newdata = NULL, ...)
```

Arguments

`fits`	A `spatial_fit` object, or a named list of them (e.g. `list(GWR = gwr_obj, Bayesian = bayes_obj)`). The names are used as the model labels and every element must have one; an unnamed list is an error, and so are duplicated names — `model` is the key the comparison table is assembled on, so two fits sharing a name cannot be told apart in the output.
`newdata`	Optional sf object for out-of-sample evaluation. Must contain the response variable and all predictors. If NULL, in-sample metrics are computed.
`...`	Extra arguments passed to predict().

Value

A data.frame with one row per model and columns for model name and all regression metrics.

Percentage errors on responses with zeros

MAPE divides by the observed value and SMAPE by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return Inf or NaN, both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. **The returned value does not record how many rows that was**, and the n column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, MAPE is an average over the 58 non-zero rows reported as though it covered all 120. SMAPE fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

See Also

Other model evaluation: compare_models(), compare_models_cv(), residual_morans_i()

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = runif(60, 0, 1000), y = runif(60, 0, 1000), a = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  pts$z <- 2 * pts$a + rnorm(60, 0, 0.3)
  fit <- fit_rf_model(pts, "z", "a", num_trees = 50, seed = 1)
  evaluate_insample(fit)                       # in-sample (out-of-bag for RF)
  evaluate_insample(fit, newdata = pts[1:20, ])  # on held-out rows
}
```

---

fitted.bayesian_fit *In-sample fitted values from a Bayesian spatial GP fit*

Description

Posterior expectation at the training locations: the column means of brms::posterior_epred(). These are **in-sample** values.

Usage

```
## S3 method for class 'bayesian_fit'
fitted(object, ...)
```

Arguments

object	A bayesian_fit.
...	Ignored.

Value

Numeric vector of length object$n (all NA if the posterior draw failed).

The result is cached

posterior_epred() is O(draws x n), and summary(), residuals(), model_metrics() and compare_models() each call fitted() independently, so the value is memoised in an environment carried in object$info$.cache (reference semantics, so it survives R's copy-on-modify). The cache holds epred column means only, which is why predict(object, summary = "median") and predict(object, type = "predict") recompute rather than reuse it. Call clear_fitted_cache if the engine has been mutated by hand after fitting.

The cache is shared by copies, and validated

An environment has reference semantics, which is what makes the memo survive R's copy-on-modify – but it also means fit2 <- fit gives the two objects *the same* cache. Assigning a different data_sf to the copy would then have returned the original's cached values, at the original's length, which residuals() silently recycled against the copy's shorter response. The entry therefore carries the n and a digest of the training data it was computed from, and is recomputed whenever either fails to match, so a copy with different data recomputes instead of reading the original's answer.

Two consequences of the shared environment remain and cannot be removed from here: clear_fitted_cache on one copy empties the cache both share (harmless – the other simply recomputes), and identical() cannot distinguish two fits by their caches. The digest covers data_sf only, not $engine: a hand-mutated brmsfit is what clear_fitted_cache is for.

---

fitted.gwr_fit *In-sample fitted values from a GWR fit*

Description

Reads the fitted values out of the GWmodel result, which stores them under one of several names depending on version and entry point; the extraction falls back through the local coefficients and the residuals when no direct column is present. These are **in-sample** values – each observation was inside its own bandwidth window – so summary() on a gwr_fit reports an optimistic fit. Use cv_gwr for a spatially blocked estimate.

Usage

```
## S3 method for class 'gwr_fit'
fitted(object, ...)
```

Arguments

object	A gwr_fit.
...	Ignored.

Value

Numeric vector of length object$n (NA where extraction failed).

---

fitted.rf_fit *Out-of-bag predictions from a random forest fit*

Description

Returns out-of-bag predictions rather than in-sample ones. See fit_rf_model for why, and for what it means for summary().

Usage

```
## S3 method for class 'rf_fit'
fitted(object, ...)
```

Arguments

object	An rf_fit.
...	Ignored.

Value

Numeric vector of length object$n.

---

fit_bayesian_spatial_model *Fit a Bayesian spatial regression with a 2D Gaussian Process (via brms)*

Description

Fits a regression whose residual spatial structure is modelled explicitly, as a Gaussian process over the coordinates, rather than left in the errors. Two things follow, and they are the reasons to reach for this backend. First, every quantity comes with a posterior, so predictions carry calibrated intervals instead of point estimates – score them with cv_bayes(), which reports held-out interval coverage and CRPS. Second, the fitted length-scale is itself an estimate of how far the spatial dependence reaches, a number you can read and report.

Usage

```
fit_bayesian_spatial_model(
  data_sf,
  response_var,
  predictor_vars,
  family = NULL,
  gp_k = NULL,
  gp_c = NULL,
  gp_iso = FALSE,
  prior = NULL,
  chains = 4,
  iter = 2000,
  warmup = floor(iter/2),
  cores = getOption("mc.cores", 1L),
  seed = 123,
  backend = c("auto", "cmdstanr", "rstan"),
  control = list(),
  compute_loo = TRUE,
  standardize_predictors = FALSE,
  check_convergence = TRUE,
  pointize = "auto",
  boundary = NULL,
  .already_prepped = FALSE
)
```

Arguments

`data_sf`	An sf object with response, predictors, and geometries.
`response_var`	Response column name.
`predictor_vars`	Predictor column names. May be `character(0)` for an intercept-only model: the response is then explained by the intercept and the spatial Gaussian process alone, which is the right baseline for asking how much of the surface is spatial structure rather than covariate effect (and the natural null model for comparing against a covariate model with `compare_models`).
`family`	A model family accepted by `brms::brm()` (a stats family function or a brms family object). Default NULL (resolved to `stats::gaussian()`).
`gp_k`	Positive integer giving the number of GP basis functions *per dimension*, or NULL (default) to derive it from the length-scale/domain ratio. Note that the fitted model carries `gp_k^2` basis functions, not `gp_k` (see Details).
`gp_c`	Positive numeric boundary factor for the approximate GP, or NULL (default) to derive it alongside `gp_k`. The boundary must be wide enough to contain the longest plausible correlation range; a value that is too small truncates the domain and degrades the approximation for smooth, long-range surfaces.
`gp_iso`	Logical; passed to `brms::gp(iso = )`. `FALSE` (the default) fits a separate length-scale per coordinate axis, letting the model learn any directional structure from

the data. TRUE fits a single shared length-scale, which – because the coordinates are standardised per axis beforehand – makes the kernel anisotropic in the original CRS by whatever ratio sd(X)/sd(Y) happens to take. See Details.

prior — Optional brms prior specification. When NULL and standardize_predictors = TRUE, weakly informative normal(0, 5) priors are set on regression coefficients. A data-informed GP length-scale prior is always appended automatically unless prior already contains an entry with class = "lscale".

chains — Number of MCMC chains. Default 4.

iter — Total iterations per chain. Default 2000.

warmup — Warmup iterations. Default floor(iter/2).

cores — Number of cores for the sampler, one chain per core. Default getOption("mc.cores", 1L) – the same convention **brms** uses itself, so options(mc.cores = 4) once per session runs the four default chains in parallel everywhere. The previous default of detectCores() - 1 took every core but one on any machine, which is not what a shared server or a check farm wants.

seed — Integer seed. Default 123.

backend — "auto" (default), "cmdstanr", or "rstan". "auto" uses **cmdstanr** only when a CmdStan build is actually available – the **cmdstanr** package is a thin interface and can be installed without one (cmdstanr::install_cmdstan() builds it) – and **rstan** otherwise, which **brms** always brings. An explicit "cmdstanr" with no usable CmdStan is an error that says how to install it, rather than a failure from inside the sampler.

control — Named list of sampler controls, *merged* over the package defaults list(adapt_delta = 0.9, max_treedepth = 12) rather than replacing them. Passing list(max_treedepth = 15) therefore keeps adapt_delta = 0.9 – which matters, because that is exactly the setting the divergence warning tells you to raise.

compute_loo — Logical; compute PSIS-LOO. Default TRUE.

standardize_predictors — Logical; center and scale numeric predictors before fitting. Default FALSE. When TRUE, the scaling parameters are stored in the return value so predictions can be computed correctly.

check_convergence — Logical; after fitting, check for divergences, low ESS, and high R-hat and issue warnings. Default TRUE.

pointize — Strategy for non-point geometry coercion.

boundary — Optional polygonal sf/sfc for CRS harmonization.

.already_prepped — Logical (internal). If TRUE, skip the prep_model_data() call because the caller has already projected, coerced, and filtered the data. The data must then have plain POINT geometry (an error is raised otherwise). Used by the CV internals to avoid a redundant second pass on every fold. End users should leave this at the default FALSE.

Details

Choose it over `fit_gwr_model()` when you want one global relationship plus an explicit spatial random field, and uncertainty you can defend; choose GWR instead when the question is how a coefficient *varies* across the map. Choose `fit_rf_model()` when predictive accuracy matters more than an interpretable model and the response is non-linear in the predictors. The cost here is time: this is full MCMC via 'brms' and Stan, so it is minutes rather than seconds, and the GP is fitted through a reduced-rank basis approximation whose size (`gp_k`) trades fidelity against runtime.

GP basis count and boundary factor. `brms::gp()` builds a full tensor grid over its covariates, so a term `gp(..x, ..y, k = gp_k)` carries `gp_k^2` basis functions – the `gp_k` argument is the count *per dimension*, not the total rank. Both `gp_k` and `gp_c` are therefore chosen from the ratio of the estimated length-scale to the domain extent, following Riutort-Mayol et al. (2023), rather than from the number of observations: `gp_c` is set large enough to contain the upper length-scale bound, and `gp_k` large enough to resolve the lower one. The derived value is typically 21-25 per dimension and is largely independent of `n`.

The domain extent used is the one `brms::gp(c = )` itself multiplies: the full pooled range of the column-centred coordinates (`brms:::choose_L()`, taken over the **unique** coordinate rows, because `brms:::.data_gp()` reduces the covariates to unique rows first under the default `gr = TRUE` – so repeat visits to one location do not widen the domain), not the per-axis half-range in which Riutort-Mayol et al. state their inequalities. Both constraints are really constraints on the boundary $L = c \times S$, so expressing them in brms's units is what keeps `gp_c`, `gp_k` and `$info$gp_ell_min` describing the basis brms actually builds. A `gp_c` derived on the half-range convention and handed to `brms::gp()` produces a boundary twice as wide as intended, against which `gp_k` under-resolves by a factor of two.

The GP term is built with `scale = FALSE`. `brms::gp()` otherwise rescales its covariates so the maximum Euclidean distance between two points is 1, and reports `lscale` in that space; since this function already standardises the coordinates, and the length-scale prior, `gp_c` and the adequacy check below are all expressed in those standardised units, a second normalisation would leave every length-scale quantity in the wrong units.

After fitting, the posterior length-scale is compared against the smallest scale the chosen basis can resolve (`1.75 * gp_c * S / gp_k`, stored as `$info$gp_ell_min`); a warning is issued when more than 10\ which is the signal that `gp_k` should be raised.

Coordinate scaling and anisotropy. Before fitting the GP, X and Y coordinates are each centred and divided by their own standard deviation. This is a conditioning step: easting and northing frequently span very different ranges in a projected CRS, and handing Stan raw metres samples poorly.

Because the axes are scaled independently, a *single* shared length-scale in the scaled space corresponds to an anisotropic kernel in the original CRS, stretched by whatever ratio `sd(X)/sd(Y)` happens to take. That ratio is a property of how the sampling locations are laid out, not of the process being modelled, so it is not a defensible source of anisotropy.

`gp_iso = FALSE` (the default) therefore fits one length-scale per axis, letting the model estimate directional structure from the data instead of inheriting it from the standardisation. Set `gp_iso = TRUE` to recover the previous single-length-scale behaviour.

Note that `gp_iso` does not affect cost: `brms::gp()` builds a tensor grid either way, so the model carries `gp_k^2` basis functions regardless. The stored `$info$coord_scaling` list records the scaling strategy, and `$info$gp_iso` records which kernel was used.

Value

A `bayesian_fit` object (inherits from `spatial_fit`). Supports `predict()`, `fitted()`, `residuals()`, `coef()`, `summary()`, and `model_metrics()`. Model-specific metadata lives in `$info` (coords – the names of the scaled coordinate columns handed to `brms::gp()`; coord_scaling, predictor_scaling, gp_k, gp_c, gp_iso, gp_n_basis, gp_ell_min, gp_S – the pooled centred range `brms::gp(c = )` multiplies; gp_xy_range – the training extrema of the scaled coordinates, which `predict()` uses to pin the GP boundary; gp_lengthscale_bounds – the `c(lower, upper)` the length-scale prior was calibrated over; gp_lscale_prior – the length-scale prior `brms::validate_prior()` reports the model will *actually* use, which is not necessarily the one this function requested (several entries, semicolon-separated, if brms resolved the axes differently); loo, looic, convergence_ok, convergence_diagnostics). The raw brmsfit is in `$engine`.

References

Riutort-Mayol, G., Burkner, P.-C., Andersen, M. R., Solin, A. and Vehtari, A. (2023). Practical Hilbert space approximate Bayesian Gaussian processes for probabilistic programming. *Statistics and Computing* **33**, 17. doi:10.1007/s11222022101672

See Also

Other model fitting: `fit_gwr_model()`, `fit_rf_model()`, `new_spatial_fit()`, `prep_model_data()`

Examples

```
## Not run:
# Not run: fits with Stan, which needs a working C++ toolchain and takes
# minutes of MCMC -- both outside what an example may assume.
if (requireNamespace("brms", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 60
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * st_coordinates(dat)[, 1] + 2 * dat$elev + rnorm(n)
  fit <- fit_bayesian_spatial_model(dat, "price", "elev",
                                    chains = 2, iter = 500,
                                    compute_loo = FALSE)
  summary(fit)
  head(predict(fit, newdata = dat))
}

## End(Not run)
```

`fit_gwr_model` *Fit a Geographically Weighted Regression (GWR) via GWmodel*

Description

Fits a GWR using GWmodel on an sf dataset with either adaptive or fixed bandwidth.

Usage

```
fit_gwr_model(
  data_sf,
  response_var,
  predictor_vars,
  adaptive = TRUE,
  bandwidth = NULL,
  kernel = c("bisquare", "gaussian", "tricube", "boxcar", "exponential"),
  .already_prepped = FALSE
)
```

Arguments

`data_sf`	An sf object with response, predictors, and geometries.
`response_var`	Response column name.
`predictor_vars`	Predictor column names.
`adaptive`	Logical; use adaptive bandwidth. Default TRUE. When TRUE, bandwidth is an integer number of nearest neighbours. When FALSE, bandwidth is a fixed distance in CRS units.
`bandwidth`	Optional numeric bandwidth value. For adaptive mode this is an integer (number of neighbours); for fixed mode a distance in the units of the **projected** CRS the fit runs in – `prep_model_data()` projects geographic input before the bandwidth is used, so 0.2 supplied for lon/lat data is 0.2 metres, not 0.2 degrees. Read the CRS off `sf::st_crs(fit$data_sf)`, or pass `target_crs` to `ensure_projected` beforehand to fix the units yourself. If NULL (default), bandwidth is selected automatically via `GWmodel::bw.gwr()`. With `adaptive = FALSE`, a bandwidth smaller than a ten-thousandth of the data's extent raises a warning naming the extent and the CRS the fit runs in: every local window is then likely to be empty, which used to produce a fit whose coefficients were all `NaN` with nothing raised anywhere.
`kernel`	Kernel function type. One of "bisquare" (default), "gaussian", "tricube", "boxcar", "exponential".
`.already_prepped`	Logical (internal). If `TRUE`, skip the `prep_model_data()` call because the caller has already projected, coerced, and filtered the data. Used by the CV internals to avoid a redundant second pass on every fold. End users should leave this at the default `FALSE`.

Value

A gwr_fit object (inherits from spatial_fit). Supports predict(), fitted(), residuals(), coef(), summary(), and model_metrics(). Model-specific metadata lives in $info (bandwidth, adaptive, kernel, AICc, and bandwidth_is_fallback – TRUE when automatic selection failed and the arbitrary fallback was used). The raw GWmodel result is in $engine.

Collinearity diagnostics

The function computes the **scaled condition index** of the design – the ratio of the largest to the smallest singular value after each column is scaled to unit length (Belsley, Kuh & Welsch 1980) – and warns when it exceeds 30, the conventional threshold, which Wheeler & Tiefelsdorf (2005) carry over to the local designs of GWR. Scaling makes the index independent of the predictors' units; kappa() on the raw matrix is not, and a threshold on it is a threshold on nothing in particular. A **global** index is computed on the full design (intercept plus predictors). In addition, a **local** spot-check is performed at up to 30 locations – every location when there are 30 or fewer, otherwise 30 spread evenly over the extent (evenly spaced ranks of the observations ordered by x, then y), so the diagnostic is reproducible, draws no random numbers, does not depend on the row order of the data, and the count is not configurable. For each sampled point the nearest neighbours within the bandwidth window – the bandwidth the model is actually fitted with, not a stand-in – are selected and the condition number of that local design sub-matrix is evaluated. That sub-matrix is the predictors **plus an intercept column**, matching the design GWmodel fits, and is unweighted; the global condition number is computed on the predictors alone, so the two numbers are not directly comparable. An indicator that is constant inside a window is collinear with the intercept and with nothing else, which is why the intercept has to be there. A non-finite condition number counts as extreme: kappa() returns Inf for an exactly singular design, which is the worst case, not an exempt one.

A warning is issued whenever **any** sampled location has a singular or near-singular local design; the wording reports a percentage when more than 25\ R warnings, not log lines.

After the fit, the local coefficient surfaces are scanned and a further warning counts local regressions that came back non-finite – their windows were singular. fitted(), residuals(), summary() and model_metrics() all drop those rows, so when this warning fires the metrics describe only the part of the study area that fitted.

Because the local spot-check examines only a subset of locations, it may not detect every problematic neighbourhood. Users working with highly clustered data or near-collinear predictors should consider a full local-collinearity audit as a post-fit diagnostic.

See Also

Other model fitting: fit_bayesian_spatial_model(), fit_rf_model(), new_spatial_fit(), prep_model_data()

Examples

```
if (requireNamespace("GWmodel", quietly = TRUE) &&
    requireNamespace("sp", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 60
```

```
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * st_coordinates(dat)[, 1] + 2 * dat$elev + rnorm(n)
  fit <- fit_gwr_model(dat, "price", "elev", bandwidth = 30)
  summary(fit)
  head(predict(fit, newdata = dat))   # newdata is re-projected if needed
}
```

`fit_rf_model` *Fit a random forest via ranger*

Description

Fits a regression random forest on an `sf` dataset and returns it as a `spatial_fit`, so it works with `cv_spatial`, `predict_surface`, `area_of_applicability` and the `plot()` method like any other backend.

Usage

```
fit_rf_model(
  data_sf,
  response_var,
  predictor_vars,
  num_trees = 500L,
  mtry = NULL,
  min_node_size = NULL,
  importance = c("permutation", "impurity", "none"),
  include_coords = FALSE,
  seed = 123L,
  num_threads = NULL,
  .already_prepped = FALSE,
  ...
)
```

Arguments

`data_sf`	An sf object with response, predictors and geometry.
`response_var`	Response column name.
`predictor_vars`	Predictor column names.
`num_trees`	Number of trees. Default 500.
`mtry`	Predictors sampled per split. `NULL` uses ranger's default.
`min_node_size`	Minimum node size. `NULL` uses ranger's default.
`importance`	`"permutation"` (default), `"impurity"` or `"none"`. Impurity importance is biased toward continuous and high-cardinality predictors (Strobl et al. 2007), which is why permutation is the default despite costing more.

include_coords Add the coordinates as predictors. Default FALSE; see above.

seed Seed passed to ranger. Default 123.

num_threads Threads for ranger. Default NULL means getOption("mc.cores", 1L): one thread unless the session has opted in to more, the convention **brms** and **parallel** use. ranger's own default is every core on the machine, which is the wrong default for a package function (a check farm limits jobs to two cores, and cv_rf(parallel =) would multiply it by the worker count). Predictions do not depend on the thread count, only speed does; pass parallel::detectCores() to use them all.

.already_prepped Internal; skip prep_model_data() because the caller has already projected and cleaned the data.

... Passed to ranger::ranger(). ranger's own spellings of the arguments this function already sets (num.trees, min.node.size, num.threads, mtry, importance, seed, x, y) are rejected with a message naming the wrapper argument to use instead – passing them here would reach ranger() twice and fail the call.

Value

An rf_fit object (inherits from spatial_fit). $info carries num_trees, mtry, min_node_size, importance_type, importance (a named numeric, or NULL when importance = "none"), include_coords, oob_rmse and oob_r_squared (each NA_real_ when ranger did not compute it – forwarding oob.error = FALSE through ... is one way to get there), fitted_are_oob (always TRUE; summary() reads it to label its metrics) and seed. The raw forest is in $engine.

Coordinates are not predictors by default

Handing a random forest the x and y coordinates lets it reproduce the training surface almost exactly by memorising location, and then fail badly anywhere it has not seen. Random cross-validation will not catch this – nearby points leak between folds, so the memorised surface scores well – which is how the practice became common. Meyer et al. (2019) show the collapse directly. include_coords therefore defaults to FALSE, and setting it to TRUE logs a caution (it is a deliberate choice, so it is not raised as an R warning). If you do use it, score the model with cv_spatial and blocked folds, never with the out-of-bag error.

The out-of-bag error is a random hold-out

ranger's OOB error holds each observation out of the trees that did not sample it. That is a random hold-out, so under spatial autocorrelation it is optimistic for exactly the reason random k-fold is: the trees that "did not see" a point almost certainly saw its neighbours. It is reported as $info$oob_rmse and $info$oob_r_squared and labelled as OOB everywhere it appears. Use cv_rf for a spatial estimate.

What fitted() returns

fitted() on an rf_fit returns **out-of-bag** predictions, not in-sample ones, following the convention of the random forest packages themselves. In-sample predictions from a forest are close to memorisation and would make summary() report a fictitious R-squared. The consequence is

that `summary()` means something different here than for a `gwr_fit` or `bayesian_fit`, whose fitted values are in-sample: do not compare the two directly. `compare_models_cv` exists for that.

References

Meyer, H., Reudenbach, C., Wöllauer, S. and Nauss, T. (2019). Importance of spatial predictor variable selection in machine learning applications – moving from data reproduction to spatial prediction. *Ecological Modelling* 411, 108815. doi:10.1016/j.ecolmodel.2019.108815

Strobl, C., Boulesteix, A.-L., Zeileis, A. and Hothorn, T. (2007). Bias in random forest variable importance measures: illustrations, sources and a solution. *BMC Bioinformatics* 8, 25. doi:10.1186/14712105825

See Also

`cv_rf` for a spatially blocked performance estimate, `area_of_applicability`, which can take `weights = pmax(fit$info$importance, 0)`.

Other model fitting: `fit_bayesian_spatial_model()`, `fit_gwr_model()`, `new_spatial_fit()`, `prep_model_data()`

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 150
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000),
               a = rnorm(n), b = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$z <- 2 * dat$a - dat$b + rnorm(n, 0, 0.3)
  fit <- fit_rf_model(dat, "z", c("a", "b"))
  fit
  fit$info$importance
}
```

---

get_voronoi_seeds *Generate seed points for Voronoi tessellation*

Description

Creates an sf POINT layer of "seed" locations. Multiple strategies are supported: user-provided points, uniform random sampling within a boundary, or k-means clustering of a sampling cloud.

Usage

```
get_voronoi_seeds(
  boundary = NULL,
  method = c("kmeans", "random", "provided"),
  n = NULL,
  seeds = NULL,
  sample_points = NULL,
  kmeans_nstart = 10,
  kmeans_iter = 100,
  set_seed = NULL
)
```

Arguments

boundary	Optional polygonal sf object defining the sampling area.
method	One of "kmeans", "random", "provided".
n	Integer; number of seeds to return. Required for `method = "kmeans"` and `method = "random"`. **Ignored** for `method = "provided"`, where every row of `seeds` is returned; a mismatch between `n` and `nrow(seeds)` is reported as a warning. For `method = "kmeans"` it is an upper bound rather than a guarantee: k-means cannot produce more centres than there are distinct positions in the sampling cloud, nor as many centres as there are rows. When `n` exceeds either ceiling it is clamped, with a warning naming the count actually used — `n = nrow(sample_points)` is the common case, and yields `nrow(sample_points) - 1` seeds. Check `nrow()` on the result rather than assuming `n`.
seeds	sf POINT object of user-provided seeds (method = "provided").
sample_points	Optional sf POINT cloud for k-means clustering. Only the first two coordinate columns are clustered, so a Z or M dimension does not join the distance calculation and dominate it; rows with empty or non-finite coordinates are dropped with a warning rather than reaching `stats::kmeans()`, which fails on them without naming a cause. A lon/lat cloud is projected before clustering.
kmeans_nstart	Integer; nstart for kmeans(). Default 10.
kmeans_iter	Integer; iter.max for kmeans(). Default 100.
set_seed	Optional integer RNG seed.

Value

An sf POINT object with seed_id and method columns.

See Also

Other tessellation: `build_tessellation()`, `create_grid_polygons()`, `create_voronoi_polygons()`, `plot_tessellation_map()`

Examples

```
library(sf)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
get_voronoi_seeds(bnd, method = "random", n = 5, set_seed = 1)
```

---

gp_lengthscale_bounds *Heuristic length-scale bounds for a squared-exponential GP*

Description

Computes sensible prior bounds for the GP length-scale parameter ℓ of a squared-exponential (exponentiated-quadratic) kernel, $k(h) = \exp(-h^2/(2\ell^2))$. The "effective range" where correlation drops to ~5\ $\ell\sqrt{2\ln 20} \approx 2.45\,\ell$.

Usage

```
gp_lengthscale_bounds(coords_xy, q_small = 0.25, max_n = 1000L)
```

Arguments

coords_xy	Numeric matrix or data.frame of coordinates with at least two columns; the first two are used, and replicated rows are collapsed before the distance quantiles are taken — `brms::gp()` defaults to `gr = TRUE` and reduces its covariates to unique rows, so a heavily-sampled station would otherwise weight the quantile by how often it was measured rather than by where the sites are.
q_small	Numeric quantile for the lower bound, a single number in `[0, 1]`. Default 0.25. Previous versions used 0.1, but the 10th percentile can be dominated by within-cluster spacing in clustered data, producing a misleadingly small lower bound.
max_n	Maximum number of *distinct* points to use in the distance computation. Default 1000. Set to `Inf` to use all of them.

Details

Subsamples large datasets to avoid O(n^2) memory and time cost.

Value

Named numeric vector `c(lower, upper)` on the length-scale; `c(lower = 0.001, upper = 1)` when fewer than two distinct locations or no positive distances remain.

Examples

```
set.seed(1)
xy <- cbind(runif(50), runif(50))
gp_lengthscale_bounds(xy)
```

`gwr_model_selection` *Forward model selection for geographically weighted regression*

Description

Wraps `GWmodel::gwr.model.selection()`, which grows a GWR model one predictor at a time and scores every intermediate model with a corrected Akaike information criterion, and returns the results as a ranked table rather than the two loosely-coupled lists GWmodel produces.

Usage

```
gwr_model_selection(
  data_sf,
  response_var,
  candidate_vars,
  bandwidth = NULL,
  adaptive = TRUE,
  kernel = c("bisquare", "gaussian", "tricube", "boxcar", "exponential"),
  bw_approach = c("AICc", "CV"),
  max_models = 200L,
  dmat_max_n = 2000L,
  quiet = TRUE,
  .engine = .gwr_ms_engine
)
```

Arguments

`data_sf`	An `sf` object with response, predictors and geometry.
`response_var`	Response column name.
`candidate_vars`	Character vector naming at least two **numeric** predictors to choose among. Factor, character and logical candidates are refused: GWmodel fits a factor as several model-matrix columns while this sweep counts it as one variable, so the criteria would not be comparable, and `fit_gwr_model()` – the documented next step – takes only numerics. Encode them as numeric indicators first.
`bandwidth`	Bandwidth held fixed across all candidate models. If `NULL` (default) it is selected with `GWmodel::bw.gwr()` on the model containing every candidate. Integer neighbour count when `adaptive = TRUE`; otherwise a distance in the units of the **projected** CRS the sweep runs in, which `prep_model_data()` may have chosen for you – geographic input is projected before the bandwidth is used, so a value in degrees would be read as metres.
`adaptive`	Logical; adaptive (nearest-neighbour) bandwidth. Default `TRUE`.
`kernel`	Weighting kernel. One of `"bisquare"` (default), `"gaussian"`, `"tricube"`, `"boxcar"`, `"exponential"`.
`bw_approach`	Criterion for the bandwidth search: `"AICc"` (default) or `"CV"`. Matches `fit_gwr_model`'s default.

max_models	Refuse the call if the sweep would exceed this many model fits. Default 200, which admits up to 19 candidates.
dmat_max_n	Precompute and reuse an n x n distance matrix when n is at most this. Default 2000 (about 32 MB). Set to 0 to disable.
quiet	Discard GWmodel's progress output. Default TRUE, because GWmodel writes it with bare cat() – which no suppressMessages() can silence – and emits one block per candidate model, so it scales with the square of the candidate count. Set FALSE to watch a long sweep progress.
.engine	Internal; injectable backend used for testing.

Value

An object of class gwr_model_selection, a list with: best (character vector of the selected predictors); table (ranked data.frame of every model evaluated, with columns rank, n_vars, variables and criterion); criterion (label for the criterion actually read, noting when it had to be located positionally); response_var and candidate_vars (the response and the full candidate set the sweep ran over, both echoed by print()); bandwidth, bandwidth_source, adaptive and kernel (the smoothing held fixed across the sweep, and where it came from); n_obs, n_models, used_dmat; and raw (GWmodel's unmodified return: the two-element list of its model list and its diagnostic table).

What this optimises, and what it does not

The criterion is **in-sample**. AICc penalises the effective number of parameters, so it is not the same thing as maximising fit, but it is still computed on the data the model was fitted to, and under spatial autocorrelation an in-sample criterion is optimistic in a way that a spatially blocked estimate is not. Treat this as fast screening. select_features_forward performs the same forward search against a spatially blocked cross-validated score; it costs far more and is the one to trust when the answer matters. When the two disagree, the disagreement is itself informative – it usually means a candidate is predictive only locally.

Two further limitations are structural rather than incidental:

- **One bandwidth for every model.** Comparing criteria across models requires holding the smoothing fixed, but the bandwidth is itself a fitted quantity, and the value chosen for the full model is not optimal for a one-predictor model. This is how the method is defined (Lu et al. 2014); it is not an implementation shortcut. Refit the selected model with bandwidth = NULL to re-optimise once the variable set is settled.
- **The null model is never evaluated.** The sweep starts from one predictor, so the result always names at least one. It cannot tell you that none of the candidates help.

Cost

The sweep fits p * (p + 1) / 2 GWR models for p candidates – 55 at p = 10, 210 at p = 20 – each over all n locations. max_models stops the call rather than letting it run for hours.

Using $raw with GWmodel directly

raw is GWmodel's own list(model.list, GWR.df), so its two elements have to be unpacked before GWmodel's own helpers will take them: GWmodel::gwr.model.view() takes (DeVar, InDeVars, model.list), so the call is

```
GWmodel::gwr.model.view(sel$response_var, sel$candidate_vars, sel$raw[[1]])
```

– sel$raw[[1]], not sel$raw. The diagnostic table is sel$raw[[2]], an unlabelled numeric matrix whose columns are bandwidth, AIC, AICc, RSS in that order; the criterion column of $table is its third column.

References

Lu, B., Harris, P., Charlton, M. and Brunsdon, C. (2014). The GWmodel R package: further topics for exploring spatial heterogeneity using geographically weighted models. *Geo-spatial Information Science* 17(2), 85–101. doi:10.1080/10095020.2014.917453

See Also

select_features_forward for the blocked cross-validated counterpart, fit_gwr_model to fit the selected model.

Other cross-validation: area_of_applicability(), cv_bayes(), cv_gwr(), cv_rf(), cv_spatial(), estimate_sac_range(), make_folds(), select_features_forward()

Examples

```
if (requireNamespace("GWmodel", quietly = TRUE) &&
    requireNamespace("sp", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 80
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000),
               a = rnorm(n), b = rnorm(n), noise = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$z <- 2 * dat$a - dat$b + rnorm(n, 0, 0.5)
  sel <- gwr_model_selection(dat, "z", c("a", "b", "noise"), bandwidth = 30)
  sel$best
  fit <- fit_gwr_model(dat, "z", sel$best)
}
```

---

harmonize_crs *Harmonize CRS between two spatial objects*

Description

Aligns two sf objects to a common CRS.

Usage

```
harmonize_crs(
  a,
  b,
  prefer = c("a", "b"),
  target_crs = NULL,
  on_transform_error = c("stop", "set_crs")
)
```

Arguments

a, b	Objects of class sf or sfc.
prefer	Which object's CRS to keep ("a" or "b").
target_crs	Optional target CRS to apply to both.
on_transform_error	What to do when st_transform() fails: "stop" (default) raises an error immediately; "set_crs" falls back to st_set_crs() (UNSAFE — coordinates are NOT reprojected, only the CRS label is overwritten). The "set_crs" option exists only for rare edge cases where you are certain the coordinates already match the target CRS definition.

Details

When one input carries no CRS, the same lon/lat heuristic `ensure_projected()` uses decides what happens, so both entry points place identical data in the same place: coordinates that look like degrees are taken as EPSG:4326 and **reprojected** to the other object's CRS (or `target_crs`); coordinates that do not are *stamped* with `sf::st_set_crs()`, which relabels without moving them. Either way the assumption is announced with a warning.

Value

A named list with components a and b.

Examples

```
library(sf)
a <- st_as_sf(data.frame(x = c(500000, 500100), y = c(4000000, 4000100)),
              coords = c("x", "y"), crs = 32632)
b <- st_transform(a, 4326)                # same points, lon/lat
h <- harmonize_crs(a, b)                   # b is brought into a's CRS
st_crs(h$a) == st_crs(h$b)
```

make_folds *Create spatial cross-validation folds*

Description

Builds train/test splits using random K-fold, spatial block K-fold, or buffered leave-one-out strategies.

Usage

```
make_folds(
  points_sf,
  k,
 method = c("random_kfold", "block_kfold", "buffered_loo", "leave_location_out", "nndm"),
  seed = NULL,
  block_nx = NULL,
  block_ny = NULL,
  block_multiplier = 3,
  block_size = NULL,
  auto_range = FALSE,
  range_frac = 1,
  response_var = NULL,
  group_var = NULL,
  prediction_points = NULL,
  predictor_vars = NULL,
  boundary = NULL,
  buffer = NULL,
  min_train = 0.5,
  phi = NULL,
  drop_empty_blocks = TRUE
)
```

Arguments

points_sf	An sf object. Any Z or M dimension is dropped before folding: `sf::st_distance()` uses every coordinate dimension, so an XYZ layer would otherwise have elevation folded into every buffer, block and neighbour distance. CRS-less points are aligned to a `boundary` or `prediction_points` that carries a CRS — reprojected when the coordinates look like lon/lat, otherwise stamped without reprojection, warning either way.
k	Integer; number of folds. Must be a single whole number >= 1 — a fraction, `NA` or a vector is an error, because a non-integer used to truncate silently and leave the last rows in no test set at all. Required for `"random_kfold"`, `"block_kfold"` and `"leave_location_out"`; `"buffered_loo"` and `"nndm"` are leave-one-out schemes and ignore it. Not every method honours it: `"buffered_loo"` and `"nndm"` are leave-one-out schemes and always return `k = n` regardless of what was asked for, while `"block_kfold"` lowers it when the grid yields fewer than

k non-empty blocks and "leave_location_out" lowers it when there are fewer than k distinct groups. Read the k element of the returned list rather than assuming the requested value; a reduction is logged.

method — One of "random_kfold", "block_kfold", "buffered_loo", "leave_location_out" or "nndm". See **Details** for what each one does and when it is appropriate.

seed — Optional integer RNG seed.

block_nx, block_ny — Optional grid dimensions for block_kfold. Ignored when block_size or auto_range override them.

block_multiplier — Numeric, default 3. When neither block_size nor block_nx/block_ny is given, the automatic grid aims for block_multiplier * k blocks over the extent (aspect-preserving), so each fold holds out about block_multiplier blocks. With 1, every fold is one contiguous region and the score depends heavily on which region each fold happened to get; with many, the blocks shrink towards single points and the scheme drifts back towards random k-fold. 3 is a compromise between those two, not a published constant. The block size that matters for leakage is the autocorrelation range, which is what block_size and auto_range control.

block_size — Optional positive numeric minimum block edge length, **in the units of the CRS the folds are built in**. When supplied, grid dimensions are clamped so that every block is at least this wide and tall. Takes precedence over block_nx/block_ny and block_multiplier.

Which CRS that is depends on the input. Projected input is used as it stands, so block_size is in your own CRS's units. Geographic (lon/lat) input is projected first by ensure_projected(), which picks a local UTM zone or, at wide extents, an equal-area projection — a CRS you did not choose, whose units are metres but whose identity varies with the data. block_size is then interpreted in *that* CRS. The CRS actually used is recorded in params$crs of the returned list; project the data yourself before calling if you want to fix the units in advance.

A block_size in the wrong unit asks for an enormous grid, so a request above 1,000,000 blocks is refused with an error naming the grid dimensions, the extent and the CRS's units, rather than being built.

auto_range — Logical. If TRUE, the spatial autocorrelation range is estimated via estimate_sac_range() — which fits directional variograms to account for anisotropy — and used as the minimum block_size. Requires response_var. An explicit block_size takes precedence. Default FALSE. Sizing blocks from the autocorrelation range is the recommendation of Roberts et al. (2017) and what **blockCV** (Valavi et al. 2019) automates; note that **blockCV** takes the fitted variogram's range *parameter* as the block size, whereas this uses the *effective* range estimate_sac_range() returns – three times that parameter for an exponential fit – so its blocks are larger than **blockCV**'s from the same variogram.

range_frac — Passed through to estimate_sac_range() when auto_range = TRUE. A fitted range beyond the longest lag the empirical variogram was fitted over is rejected as unidentified, and block sizing falls back to geometry rather than collapsing to a single block. Default 1.0.

response_var — Character(1) response column name. Required when auto_range = TRUE.

group_var — Character(1) naming a column of points_sf that identifies the location each observation belongs to. Required for method = "leave_location_out", which keeps every observation from a location together in the same fold. Repeated measurements at the same site otherwise get split across folds, and the model is scored partly on sites it has already seen – which random k-fold reports as excellent performance.

prediction_points — Optional sf layer of the locations you actually intend to predict onto. Required for method = "nndm". The grid from predict_surface() is the natural choice; a non-POINT layer (grid cells, polygons) is reduced to representative points first, so the target distances are point-to-point rather than point-to-polygon — the latter is zero for every cell that contains a training point, which pulls the target distribution towards zero and degenerates the CV towards plain leave-one-out.

predictor_vars — Optional character vector of predictor column names. Passed to estimate_sac_range() for residual variogram estimation.

boundary — Optional polygonal sf/sfc for block_kfold.

buffer — Positive numeric distance for buffered_loo.

min_train — For method = "nndm": the smallest fraction of the data any fold's training set may be reduced to by neighbour exclusion. Default 0.5, as in CAST::nndm().

phi — For method = "nndm": the distance up to which the two nearest-neighbour distance distributions are matched, in the CRS the folds are built in; the exclusion never pushes a held-out point's nearest neighbour beyond it. In Mila et al. (2022) – and CAST::nndm() – ϕ is the autocorrelation range of the outcome: beyond it observations are effectively independent, so matching is unnecessary. estimate_sac_range() gives such a value. Default NULL = the largest prediction-to-training distance, which matches everywhere (CAST's phi = "max").

drop_empty_blocks — Logical. Default TRUE.

Details

For block_kfold, the default grid sizing is purely geometric and unrelated to the autocorrelation range of the data. When blocks are smaller than the autocorrelation range, spatially correlated observations leak across folds and CV metrics become optimistic. Use block_size to set a minimum block edge length (in CRS units), or set auto_range = TRUE to estimate the range from an empirical variogram and enforce it automatically.

Fold methods. "random_kfold" ignores geography entirely and will overstate performance on autocorrelated data. "block_kfold" separates folds geographically. "buffered_loo" holds out one point at a time and excludes everything within a fixed buffer.

"leave_location_out" groups by group_var, so all observations from a location share a fold.

"nndm" implements the distance-matching principle of Milà et al. (2022): rather than choosing a buffer arbitrarily, it sizes the exclusion around each held-out point so that the resulting training-to-test distance distribution approaches the distribution of distances from your actual prediction locations to the training data.

The procedure is the paper's own (as in `CAST::nndm()`), and it is deterministic. Let G_{ij} be the empirical distribution of prediction-to-nearest-training distances and G_j^* the distribution of each held-out point's nearest remaining training point. Starting from plain leave-one-out, the point with the smallest G_j^* at which the realised distribution exceeds the target – $G_j^*(r) > G_{ij}(r)$ – has its nearest training neighbour removed, and this repeats until no such point remains, subject to two limits: a point's nearest-neighbour distance is never pushed beyond `phi` (default: the largest prediction distance, since a training point already further than every prediction distance has nothing to match), and no fold's training set is stripped below `min_train` of the data.

The realised distribution is then never *more optimistic* than the target: $G_j^*(r) \leq G_{ij}(r)$ up to the granularity of the neighbour distances, which is the property the method exists to deliver. An earlier version of this package drew one random radius per point from G_{ij} and excluded up to the order statistic *closest* to it, which rounds down half the time: on a two-cluster layout the realised distribution exceeded the target by up to 0.17 (13\ nearest training point within 50 m against a target of 9\ *optimistic* cross-validation. `params$max_ecdf_excess` reports the largest remaining excess; compare `params$target_median` with `params$realised_median` as well.

Value

A list with method, k, folds, assignment, params. The `train`/`test` elements of each fold contain `..row_id` values (equal to row positions when the input has no pre-existing `..row_id` column), consistent with the `assignment` tibble. The returned `k` is the number of folds actually built, which is not always the `k` that was requested (see the `k` argument above), and `length(folds)` always matches it.

For the methods that work in projected space — `"block_kfold"`, `"buffered_loo"` and `"nndm"` — `params` carries a `crs` element naming the CRS the folds were built in (an `"EPSG:code"` string where there is one, otherwise the CRS's input definition). Every length in `params` — `block_size`, `sac_range`, `buffer`, `median_buffer` — is in that CRS's units, which for geographic input is a CRS `ensure_projected()` chose rather than one you passed.

Rows whose geometry is empty or has non-finite coordinates are dropped before folding, with a logged warning naming the count; they appear in no fold and in no `assignment` row.

References

Mila, C., Mateu, J., Pebesma, E. and Meyer, H. (2022). Nearest neighbour distance matching Leave-One-Out Cross-Validation for map validation. *Methods in Ecology and Evolution* **13**, 1304-1316. doi:10.1111/2041210X.13851

Roberts, D. R., Bahn, V., Ciuti, S., Boyce, M. S., Elith, J., Guillera-Arroita, G., Hauenstein, S., Lahoz-Monfort, J. J., Schroder, B., Thuiller, W., Warton, D. I., Wintle, B. A., Hartig, F. and Dormann, C. F. (2017). Cross-validation strategies for data with temporal, spatial, hierarchical, or phylogenetic structure. *Ecography* **40**, 913-929. doi:10.1111/ecog.02881

Valavi, R., Elith, J., Lahoz-Monfort, J. J. and Guillera-Arroita, G. (2019). blockCV: An R package for generating spatially or environmentally separated folds for k-fold cross-validation of species distribution models. *Methods in Ecology and Evolution* **10**, 225-232. doi:10.1111/2041210X.13107

See Also

Other cross-validation: `area_of_applicability()`, `cv_bayes()`, `cv_gwr()`, `cv_rf()`, `cv_spatial()`, `estimate_sac_range()`, `gwr_model_selection()`, `select_features_forward()`

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = runif(30, 0, 1000), y = runif(30, 0, 1000)),
  coords = c("x", "y"), crs = 32632
)
folds <- make_folds(pts, k = 3, method = "block_kfold", seed = 42)
folds$assignment          # fold membership per row
lengths(folds$folds[[1]]) # train/test row-ID splits

# Buffered leave-one-out: neighbours within 100 units excluded from training
loo <- make_folds(pts, k = 1, method = "buffered_loo", buffer = 100)
```

---

model_metrics *Compute goodness-of-fit metrics for a spatial model*

Description

Reports RMSE, MAE, MAPE, SMAPE, R^2 and adjusted R^2 for any `spatial_fit`, in one row and on one scale, so that fits from different backends can be read side by side. Reach for it to score a model on data you hold out yourself (pass it as `newdata`), or to get a quick in-sample reading of how closely a fit tracks its training data.

Usage

```
model_metrics(object, ...)

## S3 method for class 'spatial_fit'
model_metrics(object, newdata = NULL, ...)
```

Arguments

`object`	A `spatial_fit` object.
`...`	Additional arguments passed to predict().
`newdata`	Optional sf object for out-of-sample evaluation. If NULL, fitted values are used (in-sample, or out-of-bag for an `rf_fit`; see above).

Details

It is not a substitute for cross-validation. With `newdata = NULL` the numbers are in-sample for a `gwr_fit` or `bayesian_fit` – and a GWR can reach a near-perfect in-sample R^2 at a small bandwidth without predicting anything. For a figure you can report, use `cv_gwr()`, `cv_bayes()`, `cv_rf()` or `compare_models_cv()`.

Value

A data.frame with n, RMSE, MAE, MAPE, SMAPE, R2, Adj_R2. `Adj_R2` is always `NA`: GWR's effective parameter count far exceeds the global predictor count and a GP model has no simple p, so it is deliberately suppressed. A non-numeric response is an error – a character or factor response cannot be scored, and used to come back as `n = 0` with every metric `NA`; a logical response is treated as 0/1.

What the metrics are computed on

With `newdata = NULL` the metrics come from `fitted(object)`. That is **in-sample** for a `gwr_fit` or a `bayesian_fit`, but **out-of-bag** for an `rf_fit`, whose `fitted()` method returns out-of-bag predictions (see `fit_rf_model`). The returned data.frame carries no label distinguishing the two, so check `object$info$fitted_are_oob` before comparing numbers across backends – or use `compare_models_cv`, which scores every backend the same way.

Percentage errors on responses with zeros

`MAPE` divides by the observed value and `SMAPE` by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return `Inf` or `NaN`, both are averaged over the rows whose denominator is non-zero, and are `NA` when no row qualifies. **The returned value does not record how many rows that was**, and the `n` column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, `MAPE` is an average over the 58 non-zero rows reported as though it covered all 120. `SMAPE` fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

`RMSE`, `MAE` and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, `cv_bayes()` additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = runif(60, 0, 1000), y = runif(60, 0, 1000), a = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  pts$z <- 2 * pts$a + rnorm(60, 0, 0.3)
  fit <- fit_rf_model(pts, "z", "a", num_trees = 50, seed = 1)
  model_metrics(fit)                          # in-sample (out-of-bag for RF)
  model_metrics(fit, newdata = pts[1:20, ])   # on held-out rows
}
```

new_spatial_fit *Build a spatial_fit S3 object*

Description

The constructor for the spatial_fit class, and the public entry point for plugging your own model backend into this package. The three built-in fitters – fit_gwr_model(), fit_bayesian_spatial_model() and fit_rf_model() – all end by calling it, and so should a custom fit_fn written for cv_spatial(): wrapping your model in a spatial_fit is what lets it use the package's folds, metrics, comparison and area-of-applicability machinery unchanged.

Usage

```
new_spatial_fit(
  subclass,
  engine,
  formula,
  response_var,
  predictor_vars,
  data_sf,
  info = list()
)
```

Arguments

subclass	Character scalar naming the class to stamp on the object: one of the built-ins "gwr_fit", "bayesian_fit" or "rf_fit", or any name of your own for a custom backend (say "lm_fit"). It is the S3 dispatch key: predict(), fitted(), residuals() and coef() on the result all dispatch on it, so a custom subclass **requires** a matching predict.<subclass>() method to be usable with cv_spatial().
engine	The raw model object your backend produced (an lm, a ranger object, a brmsfit, ...). Nothing inspects it except your own methods.
formula	A formula.
response_var	Character(1).
predictor_vars	Character vector.
data_sf	An sf object used for fitting.
info	Named list of model-specific extras. Set fitted_are_oob = TRUE if your fitted() values are held out rather than in-sample, so summary() labels them honestly.

Details

There are two obligations. Return an object built here from your fit_fn, and define a predict() method for the subclass you chose – cv_spatial() scores folds by calling the predict() generic on the fit, so without a matching predict.<subclass>() every fold fails. A fitted.<subclass>() method returning one value per row of the fit's data_sf, in the same order, is **required** by summary() and model_metrics(): both error, naming the method to define, without one. residuals() and coef() methods are optional.

Value

An object of class `c(subclass, "spatial_fit")`.

The coef() contract

`coef()` on one of the three built-in backends either returns the coefficients or signals an error – it never returns `NULL`. A custom subclass inherits `stats::coef.default()`, which returns `NULL`, so define a `coef.<subclass>()` that errors when your backend has no coefficients; otherwise the hazard described below applies to your own fits. `coef.rf_fit()` always errors, because a forest has no coefficients; `coef.gwr_fit()` and `coef.bayesian_fit()` error when the backend cannot supply them (a missing package, an engine without the expected component). A `NULL` return would be indistinguishable from "this model genuinely has no fixed effects", so `lapply(fits, coef)` would quietly produce a shorter answer than the caller expected. Wrap in `try()` or `tryCatch()` when sweeping over a heterogeneous list of fits.

See Also

`cv_spatial()`, which consumes a custom `fit_fn`; `fit_rf_model()` for a worked built-in fitter.

Other model fitting: `fit_bayesian_spatial_model()`, `fit_gwr_model()`, `fit_rf_model()`, `prep_model_data()`

Examples

```
library(sf)
set.seed(1)
n <- 80
site <- st_as_sf(
  data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
  coords = c("x", "y"), crs = 32632
)
site$price <- 10 + 0.01 * st_coordinates(site)[, 1] + 2 * site$elev + rnorm(n)

# A custom backend: an ordinary linear model behind the spatial_fit interface.
lm_fit <- function(train_sf) {
  new_spatial_fit(
    subclass       = "lm_fit",
    engine         = lm(price ~ elev, st_drop_geometry(train_sf)),
    formula        = price ~ elev,
    response_var   = "price",
    predictor_vars = "elev",
    data_sf        = train_sf
  )
}

# Required: cv_spatial() scores each fold through the predict() generic,
# which dispatches on the subclass named above.
predict.lm_fit <- function(object, newdata = NULL, ...) {
  if (is.null(newdata)) newdata <- object$data_sf
  as.numeric(stats::predict(object$engine, st_drop_geometry(newdata)))
}
registerS3method("predict", "lm_fit", predict.lm_fit)
```

```
cv <- cv_spatial(site, "price", "elev", fit_fn = lm_fit, k = 3, seed = 1)
cv$overall
# Always check these two agree before trusting the metrics above.
c(attempted = cv$n_folds_attempted, succeeded = cv$n_folds_succeeded)
```

`plot.spatial_fit` *Plot a fitted spatial model*

Description

Diagnostic plots for a `spatial_fit`. The package previously shipped `print()` and `summary()` methods but no `plot()`, so the checks most likely to reveal a problem – is there structure left in the residuals, and where is it – had to be written by hand each time.

Usage

```
## S3 method for class 'spatial_fit'
plot(x, type = c("residuals", "observed_predicted", "variogram"), ...)
```

Arguments

`x`	A `spatial_fit`.
`type`	One of:
	`"residuals"` Residuals mapped at the training locations. Spatial structure here is the signal that the model has not captured the autocorrelation.
	`"observed_predicted"` Observed against fitted, with a 1:1 reference line.
	`"variogram"` Empirical variogram of the residuals with the fitted model overlaid, so the fit can be judged rather than trusted. The distance axis is labelled in the units of the CRS the variogram was actually fitted in, which is not necessarily the fit's own CRS (lon/lat data are projected first). A single-direction fit names its azimuth in the subtitle; a fit that did not converge says so in the caption, since the overlaid model line is then not a fit to believe. Requires 'gstat'.
`...`	Ignored.

Value

A `ggplot` object.

See Also

Other plotting: `plot_folds()`

Examples

```
# Works on any spatial_fit; a forest keeps the example free of the optional
# GWR/Stan backends.
if (requireNamespace("ranger", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  pts <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  pts$price <- 10 + 0.01 * st_coordinates(pts)[, 1] + 2 * pts$elev + rnorm(n)
  fit <- fit_rf_model(pts, "price", "elev", num_trees = 100, seed = 1)
  plot(fit, type = "residuals")
  plot(fit, type = "observed_predicted")
  if (requireNamespace("gstat", quietly = TRUE))
    plot(fit, type = "variogram")
}
```

---

plot_folds *Map a cross-validation fold scheme*

Description

Shows which fold each observation belongs to. This is the fastest way to see whether spatial blocks are actually separating the data, or whether the blocks are smaller than the autocorrelation range and therefore leaking.

Usage

```
plot_folds(folds, points_sf, boundary = NULL)
```

Arguments

folds	A list returned by make_folds().
points_sf	The sf layer the folds were built from.
boundary	Optional polygonal sf/sfc to draw underneath.

Value

A ggplot object.

See Also

Other plotting: plot.spatial_fit()

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 80
  pts <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000)),
    coords = c("x", "y"), crs = 32632
  )
  f <- make_folds(pts, k = 5, method = "block_kfold", block_size = 300)
  plot_folds(f, pts)
}
```

`plot_tessellation_map` *Plot a tessellation map with optional boundary, seeds, and features*

Description

Builds a layered ggplot2 map of polygon tessellations and optional overlays for a study boundary, seed points, and additional features.

Usage

```
plot_tessellation_map(
  tessellation_sf,
  boundary = NULL,
  seeds_sf = NULL,
  features_sf = NULL,
  fill_col = NULL,
  palette = "viridis",
  na_fill = "grey90",
  tile_alpha = 0.9,
  outline_col = "white",
  outline_size = 0.2,
  features_col = "#333333",
  features_size = 0.5,
  seeds_col = "#1f77b4",
  seeds_size = 1.5,
  boundary_col = "#111111",
  boundary_size = 0.6,
  labels = FALSE,
  label_col = "grid_id",
  label_size = 2.7,
  legend = TRUE,
  legend_title = NULL,
  theme = NULL,
  target_crs = NULL,
```

```
  title = NULL,
  subtitle = NULL,
  caption = NULL,
  xlim = NULL,
  ylim = NULL,
  expand = TRUE
)
```

Arguments

`tessellation_sf`	An sf POLYGON/MULTIPOLYGON layer. Required.
`boundary`	Optional sf/sfc polygon outline layer.
`seeds_sf`	Optional sf/sfc point layer of seed locations.
`features_sf`	Optional sf/sfc layer of additional features.
`fill_col`	Column name in tessellation_sf to map to fill. NULL = no fill.
`palette`	Viridis palette name. Default "viridis".
`na_fill`	Fill for NA values. Default "grey90".
`tile_alpha`	Alpha for filled polygons. Default 0.9.
`outline_col`, `outline_size`	Tessellation outline aesthetics.
`features_col`, `features_size`	Feature overlay aesthetics.
`seeds_col`, `seeds_size`	Seed point aesthetics.
`boundary_col`, `boundary_size`	Boundary outline aesthetics.
`labels`	Logical; draw per-cell labels. Default FALSE.
`label_col`	Column for label text. Default "grid_id".
`label_size`	Label text size. Default 2.7.
`legend`	Logical; show fill legend. Default TRUE.
`legend_title`	Optional legend title.
`theme`	A ggplot2 theme, or NULL (the default) to use `ggplot2::theme_void()`. The default is resolved inside the function rather than in the formals, so that a Suggests package is never evaluated before the `requireNamespace()` check has run.
`target_crs`	Optional CRS for plotting. When `NULL` (the default) the tessellation's own CRS is used, or — if the tessellation has none — a CRS borrowed from the first overlay that carries one, so a CRS-less grid drawn with located features still lines up. Any layer that arrives without a CRS is brought into the plot's CRS rather than dropped: reprojected when its coordinates look like longitude/latitude, otherwise stamped with a warning, since a stamp assumes the coordinates were already in that CRS.

`title`, `subtitle`, `caption`
: Plot annotations.

`xlim`, `ylim`
: Optional numeric vectors of length 2 for coordinate limits (in the plot CRS). Default NULL (auto).

`expand`
: Logical; expand plot area slightly beyond data limits. Default TRUE.

Value

A ggplot2 object.

See Also

Other tessellation: `build_tessellation()`, `create_grid_polygons()`, `create_voronoi_polygons()`, `get_voronoi_seeds()`

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = runif(20, 0, 100), y = runif(20, 0, 100)),
    coords = c("x", "y"), crs = 32632
  )
  tess <- build_tessellation(pts, method = "voronoi", quiet = TRUE)
  p <- plot_tessellation_map(tess$cells, features_sf = pts,
                             fill_col = "cell_id", legend = FALSE)
  p
}
```

`predict.bayesian_fit` *Predict from a Bayesian spatial GP model*

Description

Applies the same newdata preparation pipeline as `predict.gwr_fit()`: non-point geometries are coerced to points, the data is projected to the CRS used during fitting (via `ensure_projected()`), and rows with missing or non-finite values are dropped. Coordinate scaling and predictor standardisation stored at fit time are then applied before delegating to `brms::posterior_epred()` or `brms::posterior_predict()`.

Usage

```
## S3 method for class 'bayesian_fit'
predict(
  object,
  newdata = NULL,
  summary = c("mean", "median"),
```

```
  type = c("epred", "predict"),
  draws = FALSE,
  ...
)
```

Arguments

object	A bayesian_fit object.
newdata	An sf object with the same predictors. The response variable need not be present (true out-of-sample prediction is supported). NULL = fitted values.
summary	"mean" (default) or "median" over posterior draws.
type	"epred" (default) for expected predictions (no obs noise), or "predict" for full posterior predictive draws (includes obs noise).
draws	If TRUE, return the full posterior draw matrix instead of a point summary. Default FALSE.
...	Ignored.

Value

Numeric vector of length nrow(newdata), or a n_draws x nrow(newdata) matrix when draws = TRUE (a 1-row all-NA matrix if the posterior draw fails). With newdata = NULL the cached fitted() values are returned only for the default summary = "mean", type = "epred", draws = FALSE combination; any other combination is recomputed against the training data, because the cache holds epred column means and nothing else.

The GP boundary is pinned

brms 2.x does not store the Hilbert-space boundary L in a fitted GP basis, so brms:::.data_gp() recomputes it from whatever rows predict() is handed – which moved every eigenfunction of the approximation with the newdata bounding box while the fitted basis coefficients stayed put. Two synthetic rows at the training coordinate extrema are therefore appended before the posterior draw and dropped from the result, reproducing the boundary the model was fitted with, so chunked, fold-wise and single-call predictions agree.

That is exact only for newdata **inside** the training coordinate envelope. Beyond it the boundary has to grow whatever is done, so predictions there are extrapolation from a basis that was not built for them *and* depend on which other rows share the call – including on predict_surface()'s chunk_size. A notice is written to the log (not raised as a warning) when it happens.

---

predict.gwr_fit *Predict from a GWR spatial model*

Description

When newdata is NULL, returns the in-sample fitted values. Otherwise uses GWmodel::gwr.predict() on the new locations. newdata is first transformed to the CRS used during fitting (via ensure_projected()), so predictions are computed in a single coordinate system regardless of the CRS newdata arrives in.

Usage

```
## S3 method for class 'gwr_fit'
predict(object, newdata = NULL, ...)
```

Arguments

object	A gwr_fit object.
newdata	An sf object with the same predictors. The response variable need not be present (true out-of-sample prediction is supported). NULL = fitted values.
...	Ignored.

Value

Numeric vector aligned to nrow(newdata), with NA for rows dropped as missing or non-finite. If GWmodel::gwr.predict() fails, every value is NA and a warning says why. CRS-less newdata first receives the interpretation the training data got, so the same rows land where they did at fit time.

---

predict.rf_fit *Predict from a random forest fit*

Description

With newdata = NULL this returns **out-of-bag** predictions, not in-sample ones – see fit_rf_model.

Usage

```
## S3 method for class 'rf_fit'
predict(object, newdata = NULL, ...)
```

Arguments

object	An rf_fit.
newdata	Optional sf object carrying the same predictors. It is transformed to the CRS used at fitting time first, so a forest that includes the coordinates is not fed a different coordinate system. Categorical predictors must not carry a level the forest was never grown with – meaning a level with **no training rows**, not merely one absent from levels(): an ordinary subset, or a spatial-CV fold that holds out a whole class, keeps the unused level while the forest has no split for it. An unseen level is an error, not a guess. A predictor that was numeric or logical when the forest was grown must also arrive numeric or logical: ranger would otherwise factor-code a character column and apply the numeric split thresholds to the codes, predicting confidently from nonsense, so a character column is refused instead. (ranger sees a logical as 0/1, so either form is accepted for one.)

`...` Passed to `ranger`'s predict method. Arguments that make `ranger` return a matrix rather than one value per row – `predict.all = TRUE`, `type = "quantiles"`, `type = "se"` with `predict.all` – are rejected, because this method's contract is one number per row of `newdata`. Call `predict(fit$engine, data = ...)` directly for those. `seed` defaults to a constant rather than being left unset: an unset `seed` makes `ranger` draw one uniform from the global RNG stream per call, so the number of `predict()` calls a script happens to make (via `predict_surface`'s `chunk_size`, say) would otherwise shift every later random draw. It does not affect a regression forest's predictions; pass your own if you need one.

Value

Numeric vector, aligned to `nrow(newdata)` with `NA` for rows dropped as incomplete.

`predict_surface` *Predict a fitted spatial model onto a regular grid*

Description

Builds a prediction surface over the extent of the training data (or over a grid you supply), predicts in chunks, and returns an `sf` layer.

Usage

```
predict_surface(
  object,
  grid = NULL,
  cell_size = NULL,
  n_cells = 10000L,
  boundary = NULL,
  covariates = NULL,
  chunk_size = 5000L,
  se = FALSE,
  ...
)
```

Arguments

`object` A `spatial_fit` (e.g. from `fit_gwr_model()` or `fit_bayesian_spatial_model()`).

`grid` Optional `sf` POINT layer to predict onto. When `NULL`, a regular grid is built over the training extent. Must have at least one row. It is brought into the fit's CRS first: a CRS-less grid is given the interpretation the training data got (the assumption recorded on the fit), with a warning, and then reprojected – otherwise a CRS-less grid can land thousands of kilometres from the covariates and every cell takes the same nearest feature.

cell_size	Grid resolution in CRS units. Ignored when grid is supplied; when NULL, derived from n_cells. A value that would produce more than 5,000,000 cells is refused, naming the implied count and the CRS units – the usual cause is a value in the wrong unit. A cell_size wider than the extent yields a single centred cell.
n_cells	Approximate cell count used to derive cell_size. Default 10000. Must be a single positive finite number and at most 5,000,000; anything else is an error. Also ignored when grid is supplied – the grid you pass is used verbatim.
boundary	Optional polygonal sf/sfc; grid points outside it are dropped. Put through the same CRS replay and reprojection as grid.
covariates	Optional sf layer carrying the model's predictors. Required when the model has predictors and grid does not already contain them. Values are taken from the nearest feature.
chunk_size	Rows per prediction call. Default 5000. A pure performance knob for the GWR and random-forest backends, whose rows do not interact. For a bayesian_fit it is also that, *provided* the grid stays inside the training extent – beyond it the GP boundary has to grow and predictions depend on which rows share the call; see predict.bayesian_fit.
se	Logical; also return a standard-error/posterior-SD column where the backend supports it. Default FALSE.
...	Passed to predict().

Details

predict() on a spatial_fit requires newdata to be constructed by hand, which makes the most common downstream task – produce a map – more work than it should be. This wraps the grid construction, covariate join, chunking and CRS handling.

Prediction over a grid is embarrassingly parallel in the sense that rows do not interact, so it is chunked: for bayesian_fit the posterior draw matrix is n_draws x n_newdata, which will exhaust memory on a fine grid long before the fit itself would.

Value

An sf POINT layer with a .pred column (and .pred_se when se = TRUE and available). For an auto-generated grid the resolution is attached as attribute "cell_size". For a user-supplied grid it is only whatever "cell_size" attribute that object already carried – usually NULL, and NULL for certain if the grid had to be re-projected, since st_transform() does not preserve custom attributes. The resolution of a grid you built is not this function's to infer.

Examples

```
# Any spatial_fit works here; a forest keeps the example free of the
# optional GWR/Stan backends.
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  pts <- st_as_sf(
```

```
  data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
  coords = c("x", "y"), crs = 32632
)
pts$price <- 10 + 0.01 * st_coordinates(pts)[, 1] + 2 * pts$elev + rnorm(n)
fit  <- fit_rf_model(pts, "price", "elev", num_trees = 100, seed = 1)
surf <- predict_surface(fit, n_cells = 500, covariates = pts)
surf[".pred"]
# Check where that surface is extrapolating before mapping it.
area_of_applicability(surf, model = fit)
}
```

`prep_model_data` *Prepare and sanitize an sf dataset for spatial modeling*

Description

Ensures point geometry, projected CRS, and removes rows with missing or non-finite values in modeling columns – *including* rows whose geometry is empty or whose coordinates are not finite, which no model backend can use. All non-POINT geometries (including MULTIPOINT) are coerced to representative points via `coerce_to_points()`, so downstream coordinate extraction always aligns one row per observation.

Usage

```
prep_model_data(
  data_sf,
  response_var,
  predictor_vars,
  boundary = NULL,
 pointize = c("auto", "surface", "point_on_surface", "centroid", "line_midpoint",
    "bbox_center"),
  require_response = TRUE
)
```

Arguments

`data_sf`	An sf object.
`response_var`	Response variable column name.
`predictor_vars`	Predictor column names. May be `character(0)` for an intercept-only model (`fit_bayesian_spatial_model()` supports one; the GWR and random-forest backends do not and reject it themselves).
`boundary`	Optional sf/sfc for CRS alignment.
`pointize`	Strategy for non-point geometry coercion, passed to `coerce_to_points`. One of `"auto"` (default), `"centroid"`, `"point_on_surface"`, `"surface"` (an alias for `"point_on_surface"`), `"line_midpoint"` or `"bbox_center"`.
`require_response`	Logical; if FALSE the response column is not required to be present (useful for out-of-sample prediction where the response is unknown). Default TRUE.

Details

The response may not appear in `predictor_vars`. Using it as its own predictor is leakage no backend catches – an out-of-bag R^2 near 1 in the random forest, a silently reduced design matrix in GWR, duplicated rows and a phantom <none> entry in the GWR selection table – so it is refused here.

Column names must be syntactically valid R names (`make.names(x) == x`). Every backend builds a model formula from these names, and a name R parses as an expression – `"B5-B4"` is `B5 - B4`, `"log(a)"` is a function call – would fit a different model from the one requested while the fit object still recorded the name you asked for. Rename the column (for example with `make.names()`) before fitting.

Value

An sf object with POINT geometry, cleaned of rows carrying missing or non-finite values in the modelling columns or in the coordinates. The CRS is projected whenever one can be established. A CRS-less layer is decided by the lon/lat heuristic (see `ensure_projected`): if its bounding box fits the lon/lat envelope *and* it spans more than one unit on some axis — or carries decimal-degree-like precision — it is read as EPSG:4326 and projected, with a warning. A small planar survey inside that envelope is included in that, deliberately; only coordinates the heuristic declines are passed through as-is. Set the CRS on `data_sf` if the data are planar.

See Also

Other model fitting: `fit_bayesian_spatial_model()`, `fit_gwr_model()`, `fit_rf_model()`, `new_spatial_fit()`

Examples

```
library(sf)
dat <- st_as_sf(
  data.frame(x = 1:5, y = 5:1,
             resp = c(1, 2, NA, 4, 5),
             pred = c(1, 2, 3, 4, Inf)),
  coords = c("x", "y"), crs = 32632
)
prep_model_data(dat, "resp", "pred")  # drops rows 3 (NA) and 5 (Inf)
```

---

print.aoa *Print an area-of-applicability result*

Description

Summarises where the model may be trusted: how many prediction locations fall inside the area of applicability and how many outside, the dissimilarity threshold that separated them, and the predictors the index was computed over (naming any dropped for having no usable variance). The proportion outside is the headline number – a map that extrapolates over much of its extent is reporting predictions its training data cannot support, whatever the cross-validation score said.

Usage

```
## S3 method for class 'aoa'
print(x, ...)
```

Arguments

x	An aoa object.
...	Ignored.

Value

x, invisibly.

---

print.gwr_model_selection

Print a GWR model selection result

Description

Shows the forward-selection trail: the response, the candidate predictors, and the top-ranked models with their criterion values, so you can see both which model won and by how much. A shallow gap between the first few rows means the ranking is not well identified and the choice of predictors should not be treated as settled – worth checking before reporting one model as the selected one.

Usage

```
## S3 method for class 'gwr_model_selection'
print(x, n = 10L, ...)
```

Arguments

x	A gwr_model_selection object.
n	Number of top-ranked models to show. Default 10.
...	Ignored.

Value

x, invisibly.

print.rf_fit *Print a random forest fit*

Description

Shows the forest's shape – formula, n, number of trees, `mtry`, node size – along with the out-of-bag error and, prominently, whether the coordinates were used as predictors. That last line is the one to check: a forest fitted with `include_coords = TRUE` can memorise location and score well out-of-bag while failing everywhere it has not been.

Usage

```
## S3 method for class 'rf_fit'
print(x, ...)
```

Arguments

`x`	An `rf_fit`.
`...`	Ignored.

Value

`x`, invisibly.

print.sac_range *Print a spatial autocorrelation range*

Description

Prints the effective range as a plain number, with the directional fit summarised beneath it when one is available.

Usage

```
## S3 method for class 'sac_range'
print(x, ...)
```

Arguments

`x`	An object of class `sac_range`.
`...`	Ignored.

Value

`x`, invisibly.

print.spatial_fit *Print a fitted spatial model*

Description

Shows the one-screen summary of a `gwr_fit`, a `bayesian_fit` or a custom subclass: backend, formula, number of observations, CRS, and the few backend-specific numbers worth seeing immediately (GWR bandwidth, GP basis size). An `rf_fit` has its own method – see `print.rf_fit` – which shows the same header plus the forest settings. It is what you get by typing the object's name, and the quickest way to confirm a fit used the data, predictors and CRS you meant. For fit quality use `model_metrics()` or `summary()` instead – nothing printed here is an out-of-sample score.

Usage

```
## S3 method for class 'spatial_fit'
print(x, ...)
```

Arguments

`x`	A `spatial_fit` object.
`...`	Ignored.

Value

`x`, invisibly (called for its side effect).

residuals.bayesian_fit *In-sample residuals from a Bayesian spatial GP fit*

Description

Observed response minus `fitted.bayesian_fit` (the cached posterior expectation), so these are **in-sample** residuals.

Usage

```
## S3 method for class 'bayesian_fit'
residuals(object, ...)
```

Arguments

`object`	A `bayesian_fit`.
`...`	Ignored.

Value

Numeric vector of length `object$n`.

`residuals.gwr_fit` *In-sample residuals from a GWR fit*

Description

Observed response minus `fitted.gwr_fit`, so these are **in-sample** residuals.

Usage

```
## S3 method for class 'gwr_fit'
residuals(object, ...)
```

Arguments

`object`	A `gwr_fit`.
`...`	Ignored.

Value

Numeric vector of length `object$n`.

`residuals.rf_fit` *Out-of-bag residuals from a random forest fit*

Description

Observed response minus `fitted.rf_fit`, which for a forest is the **out-of-bag** prediction – each observation predicted only by the trees that did not see it. These are therefore already held-out residuals, unlike `residuals.gwr_fit()` and `residuals.bayesian_fit()`, which are in-sample. Feed them to `residual_morans_i()` to test whether spatial structure the forest failed to capture is still sitting in the residuals. Out-of-bag is not a substitute for spatial CV: use `cv_rf()` for an honest map-accuracy figure.

Usage

```
## S3 method for class 'rf_fit'
residuals(object, ...)
```

Arguments

`object`	An `rf_fit`.
`...`	Ignored.

Value

Numeric vector of length `object$n`.

`residual_morans_i` *Compute Moran's I on the residuals of a fitted spatial model*

Description

Given a `spatial_fit` object, extracts the residuals and the observation coordinates, builds a spatial weight matrix, and computes Moran's I together with its analytical expectation and variance under a stated null. A z-score and two-sided p-value are provided so the caller can assess whether statistically significant spatial autocorrelation remains after fitting.

Usage

```
residual_morans_i(
  fit,
  alternative = c("two.sided", "greater", "less"),
  weights = NULL,
  k = 8L,
  null = c("auto", "randomisation", "residual")
)
```

Arguments

`fit`	A `spatial_fit` object (from `fit_gwr_model`, `fit_bayesian_spatial_model` or `fit_rf_model`).
`alternative`	Character: `"two.sided"` (default), `"greater"` (positive autocorrelation), or `"less"`.
`weights`	Optional user-supplied n x n weight matrix — a base matrix or a **Matrix**-package matrix (e.g. a sparse dgCMatrix). When `NULL` (the default), a row-standardised k-nearest-neighbour weight matrix (k = 8) is built from the observation coordinates. Ties at the k-th distance – every regular grid, every site with repeat visits – share that slot's weight equally rather than being broken by row order or by which backend found them, so the matrix is a function of the geometry alone; on distinct, untied coordinates it equals spdep's `knearneigh()` + `nb2listw(style = "W")` exactly. If a non-row-standardised matrix is supplied (i.e. rows do not all sum to 1), the Cliff & Ord variance formula is still valid for general W and the computation proceeds; a note is logged (not raised as an R warning) because the magnitude of I is not directly comparable to results obtained with row-standardised weights. A `weights` argument that is not an n x n matrix is an error.
`k`	Integer number of nearest neighbours used when building the default weight matrix (ignored when `weights` is supplied). Default 8. `k >= n - 1` is a complete graph – $n(n-1)$ weights however they are stored – and is refused above 20 million of them.

`null` Which null distribution the expectation, variance and p-value are computed against. One of:

`"auto"` **(default)** `"residual"` when the design matrix can be rebuilt *and* the fit's residuals are the OLS residuals on it, `"randomisation"` otherwise.

`"randomisation"` Always the exchangeable moments.

`"residual"` Always the Cliff & Ord residual moments. Falls back to `"randomisation"` with a logged warning if the design cannot be rebuilt, and warns (but proceeds) if the residuals are not the OLS residuals on it, in which case the moments are approximate.

Both `"auto"` and `"residual"` also fall back to `"randomisation"` when the residual degrees of freedom $n - p$ are below 4 (the residual variance divides by $(n - p)(n - p + 2)$ and the normal approximation means nothing there); `"residual"` logs a warning when it does, `"auto"` does not, and `df` in the result is then $n - 1$. Read `null` in the result rather than assuming. See **Which null, and when it is approximate** above.

Details

By default, weights are constructed as a k-nearest-neighbour (k = 8) binary matrix, row-standardised. Users may supply their own weight matrix via the `weights` argument.

Value

A list with components:

observed Numeric scalar, Moran's I statistic.

expected Expected I under the null named by `null`: $-1/(n-1)$ for `"randomisation"`, $(n/S_0)\mathrm{tr}(MW)/(n-p)$ for `"residual"`.

sd Standard deviation of I under that same null.

z Standardised z-score, $(I - E[I])/sd(I)$.

p_value Two-sided (or one-sided) p-value from the normal approximation.

n Number of observations used.

null The null actually used, `"randomisation"` or `"residual"` — check this rather than assuming, since `"auto"` chooses per fit and `"residual"` can fall back.

df Residual degrees of freedom behind the moments: $n - p$ for `"residual"` (where p is the rank of the design matrix), $n - 1$ for `"randomisation"`.

Returns `NULL` with a warning if computation fails (e.g. fewer than 4 valid residuals).

Which null, and when it is approximate

Two nulls are available, and the one actually used is reported back in the `null` element of the return value.

`"randomisation"` is the classical exchangeable null: $E[I] = -1/(n - 1)$ with the Cliff & Ord randomisation variance, conditioning on the observed kurtosis. These are the moments of I for a vector whose elements are equally likely in any order.

Model residuals are not exchangeable. They are orthogonal to the design matrix, which pushes $E[I]$ materially below $-1/(n-1)$ whenever the covariates are spatially smooth — and pushes it further the more covariates there are. In a simulation with $n = 120$, six smooth covariates and *independent* errors (so the truth is "no residual autocorrelation"), OLS residuals had mean $I = -0.031$ against the exchangeable $E[I] = -0.008$; the z-score averaged -0.54 with $sd = 0.90$ instead of 0 and 1. The cost is power, which is the point of the test: at a moderate residual autocorrelation the exchangeable null rejected 13\

`"residual"` therefore uses the Cliff & Ord (1981) sec. 8.3 moments for regression residuals, with $M = I - X(X'X)^{-1}X'$ rebuilt from `predictor_vars` and `data_sf`:

$$E[I] = (n/S_0)\,\mathrm{tr}(MW)/(n-p)$$

$$Var[I] = (n/S_0)^2[\mathrm{tr}(MWMW') + \mathrm{tr}((MW)^2) + (\mathrm{tr}MW)^2]/[(n-p)(n-p+2)] - E[I]^2$$

These assume normal errors rather than conditioning on the observed kurtosis. On the simulation above they restored the z-score to mean -0.09, $sd = 1.03$, and the rejection rate to 4.3\ nominal 5\ precision.

These moments are exact for $e = My$ and for nothing else, so `null = "auto"` does not guess from the fit's class: it rebuilds `X`, regresses the response on it, and uses the residual moments only when the supplied residuals *are* those OLS residuals to numerical tolerance. A GWR wide enough to have collapsed to global OLS passes that test; the same GWR at a working bandwidth does not.

For the flexible backends neither null is exact, and `"auto"` leaves them on `"randomisation"` because forcing the OLS moments on them measurably makes matters worse, not better. Measured on null data ($n = 120$, three smooth covariates, independent errors; nominal 5\ one-sided):

backend	randomisation	residual
OLS	0.035	0.060
random forest	0.128	0.200
GWR	0.000	0.000

The random forest is anticonservative under both. Its residuals here are the **out-of-bag** ones (`residuals.rf_fit()`), not in-sample fits – they are inflated rather than shrunk (measured sd 1.08 against a true 1.00) – but they are not a linear projection of the response and they are spatially heteroscedastic, so neither set of moments describes their null distribution and the variance is understated whichever is used ($sd(z) \approx 1.3$). GWR is conservative under both, because it removes far more structure than a rank-p projection does. Treat the p-value from those backends as a rough indicator, and prefer spatially-blocked cross-validated residuals or an explicit spatial covariance model when the answer has to carry weight.

A permutation null was considered and rejected: permuting the residual vector destroys exactly the orthogonality that causes the bias, so its mean is the exchangeable $-1/(n-1)$ by construction (measured: -0.00840 against $-1/(n-1) = -0.00840$) and it reproduces the randomisation null rather than correcting it.

References

Cliff, A. D. and Ord, J. K. (1981) *Spatial Processes: Models and Applications*. Pion, London. Section 8.3.

See Also

Other model evaluation: compare_models(), compare_models_cv(), evaluate_insample()

Examples

```
# Works on any spatial_fit; a forest keeps the example free of the optional
# GWR/Stan backends.
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  dat <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000), elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  # A strong east-west trend the predictor cannot explain: the residuals
  # should still carry spatial structure, and this is what detects it.
  dat$price <- 10 + 0.02 * st_coordinates(dat)[, 1] + 2 * dat$elev + rnorm(n)
  fit <- fit_rf_model(dat, "price", "elev", num_trees = 100, seed = 1)
  # I = 0.64, z = 15.5: strong positive residual autocorrelation, exactly
  # as constructed.  A z near 0 with a large p-value would be the opposite
  # verdict -- no structure the model failed to capture.
  residual_morans_i(fit)
}
```

---

select_features_forward

Greedy forward feature selection with spatially blocked inner folds

---

Description

Selects predictors by repeatedly adding whichever candidate most improves a cross-validated score, stopping when no candidate improves it by more than `tol`.

Usage

```
select_features_forward(
  train_sf,
  response_var,
  candidate_vars,
  fit_fn,
  k = 5,
  method = c("block_kfold", "random_kfold"),
  block_size = NULL,
  metric = c("RMSE", "MAE", "R2"),
  tol = 0,
  max_vars = NULL,
  max_fits = 5000L,
```

```
  seed = 123,
  quiet = FALSE
)
```

Arguments

train_sf	Training data (sf).
response_var	Character(1).
candidate_vars	Character vector of predictors to choose among.
fit_fn	A function (train_sf, predictor_vars) returning a spatial_fit. Note the two-argument signature: selection needs to refit with different predictor sets.
k	Inner fold count. Default 5.
method	Inner fold method. Default "block_kfold".
block_size	Passed to make_folds(); inherit the outer block size so inner and outer blocks are on the same spatial scale.
metric	Score to optimise: "RMSE", "MAE" (minimised) or "R2" (maximised). Default "RMSE".
tol	Minimum improvement required to accept a variable. Default 0, meaning any improvement is accepted. The first variable is judged against the null (intercept-only) model, so tol bites from step 1 — but only when that null model can be scored. Backends that refuse a zero-length predictor_vars (fit_rf_model and fit_gwr_model both do) have no null score, and there the first variable is accepted unconditionally.
max_vars	Optional cap on how many predictors to select.
max_fits	Abort if the sweep would exceed this many model fits. Default 5000.
seed	RNG seed. It governs both the inner fold construction and the cross-validation itself: it is forwarded to cv_spatial(seed =), which draws one RNG stream per fold from it, so it also seeds the *learner* inside every fold. A stochastic fit_fn is therefore reproducible from this one value.
quiet	Logical; suppress this function's progress message()s. It does not silence R warnings, nor the package's console log echo (see spatialkit_quiet for that). Default FALSE.

Details

The inner folds must be spatial, and that is the entire point. Nested selection is only worth doing if the inner loop is blocked the same way the outer one is. Random inner folds inside blocked outer folds select variables that look predictive only because nearby points leak between train and test – and the outer loop then reports honest-looking numbers for a dishonestly chosen feature set, which is worse than not selecting at all, because the dishonesty is now hidden behind a defensible-looking validation. method therefore defaults to "block_kfold" and logs a loud caution if set to "random_kfold" (a deliberate choice, so it is not raised as an R warning).

Call this *inside* the fit_fn you pass to cv_spatial(). .cv_fit_one_fold() calls fit_fn(train_sf) on the training slice only, so anything done inside it is automatically nested and leak-free; no extra plumbing is needed. Note the cost: a sweep over p candidates costs roughly p^2 / 2 * k model fits, and nesting that inside n outer leave-one-out folds multiplies it by n. max_fits guards against that.

Value

A list with `selected` (the chosen predictors, in the order they were added), `score` (their cross-validated `metric`), `history` and `params`. `history` is a data.frame with `step`, `variable` and `score`, holding every candidate evaluated at every step; when the null model could be scored it also carries a `step = 0` row named `"<none>"` giving that baseline, so the first variable's gain can be read off directly.

See Also

Other cross-validation: `area_of_applicability()`, `cv_bayes()`, `cv_gwr()`, `cv_rf()`, `cv_spatial()`, `estimate_sac_range()`, `gwr_model_selection()`, `make_folds()`

Examples

```
if (requireNamespace("GWmodel", quietly = TRUE) &&
    requireNamespace("sp", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  pts <- st_as_sf(
    data.frame(x = runif(n, 0, 1000), y = runif(n, 0, 1000),
               a = rnorm(n), b = rnorm(n), noise = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  pts$resp <- 3 * pts$a + 2 * pts$b + rnorm(n, 0, 0.3)

  fit_fn <- function(tr, vars) fit_gwr_model(tr, "resp", vars, bandwidth = 30)
  sel <- select_features_forward(pts, "resp", c("a", "b", "noise"), fit_fn,
                                 k = 3, quiet = TRUE)
  sel$selected
}
```

`spatialkit_quiet` *Quieten (or restore) spatialkit's console log*

Description

The package writes an INFO+ trace to a session temp file (logger appender index 1) and echoes WARN+ to the console (index 2). Both `logger::log_appender()` and `logger::log_threshold()` default to `index = 1`, so the obvious two-line recipe silences the file and leaves the console untouched – which is the opposite of what anyone wants. This helper names the right index.

Usage

```
spatialkit_quiet(quiet = TRUE)
```

Arguments

quiet	TRUE (default) silences the console echo; FALSE restores the package default, WARN+. A **logger** threshold (logger::ERROR, or the value a previous call returned) sets that level instead, which is how to put back exactly what was in force rather than the default: old <- spatialkit_quiet(); spatialkit_quiet(old).

Details

Note that these are log records, not R conditions: suppressWarnings() and tryCatch(warning =) do not see them. Conditions the package raises as real R warnings are unaffected by this function.

Value

Invisibly, the threshold that was in force before the change – a **logger** level that can be passed back as quiet.

Examples

```
old <- spatialkit_quiet()      # console echo off
spatialkit_quiet(old)          # back to whatever it was
spatialkit_quiet(FALSE)        # or back to the WARN+ default
```

---

summarize_by_cell *Summarize features by polygon/cell ID*

Description

Aggregates an sf point dataset into one row per cell. By default computes counts and means, but the aggregation function is configurable.

Usage

```
summarize_by_cell(
  assigned_points_sf,
  response_var = NULL,
  predictor_vars = NULL,
  id_col = "poly_id",
  agg_funs = list(mean = function(x) mean(x, na.rm = TRUE)),
  cells_sf = NULL,
  deff = 1,
  sac = NULL,
  deff_max_n = 500L,
  quiet = TRUE
)
```

Arguments

`assigned_points_sf`
: An sf object with a cell identifier column.

`response_var`
: Optional response column name for per-cell aggregation.

`predictor_vars`
: Optional predictor column names for per-cell aggregation.

`id_col`
: Preferred name of the polygon/cell ID column.

`agg_funs`
: Named list of aggregation functions. Default `list(mean = \(x) mean(x, na.rm = TRUE))`. Additional common options: `median`, `sum`, `sd`.

`cells_sf`
: Optional polygon sf layer to join cell geometries onto the output. When supplied, the return value is an sf object with the polygon geometry from cells_sf, with one row per cell in `cells_sf` — cells that no feature fell in are kept, with `NA` summaries. Duplicate ID values in `cells_sf` would multiply those rows, so they are reported with a warning. When NULL (default), a plain data.frame/tibble is returned (previous behaviour).

`deff`
: Design-effect adjustment for standard errors. One of:

  `1` **(default)** No adjustment; the classic IID standard error `sd / sqrt(n)`, which assumes the observations within a cell are independent. No `"deff_applied"` attribute is attached, so a result carrying none was computed this way.

  `"variogram"` Compute a per-cell design effect from a fitted variogram: for `n` points in a cell with correlation matrix `R`, the effective sample size of the mean is `n^2 / sum(R)`, so `deff = sum(R) / n`. This generalises Kish – substituting a constant off-diagonal correlation recovers `1 + (n - 1) * rho` exactly – but lets correlation decay with distance, which matters increasingly as cells get larger and Kish's single-`rho` assumption degrades. Supply the fit via `sac`, or it is estimated when `response_var` is given and 'gstat' is available. Exponential, spherical and Gaussian models are supported, with a nugget and with several structured components (each weighted by its partial sill); a model of any other family falls back to `deff = 1` with a warning naming it.

  `"kish"` Estimate per-variable-type intra-class correlations (ICCs) from the grouped data using a one-way random-effects ANOVA decomposition — one ICC for the response variable and a separate ICC for the predictor variables — then apply Kish's formula per cell: `deff_i = 1 + (n_i - 1) * rho`. The ICC is the ANOVA (method-of-moments) estimator with Donner's `n0` for unequal cell sizes, not the REML estimate a mixed model returns: on a single unbalanced draw the two can differ by 0.1–0.2 (one check with cell sizes 3 to 77 and a true ICC of 0.5 gave 0.33 against REML's 0.51), while balanced designs agree to about 0.01. Neither is wrong, so do not read the difference as a defect. When multiple columns are pooled for a single ICC estimate (e.g. several predictor variables), each column is z-scored before pooling so that variables with different scales contribute equally to the variance decomposition. The response-specific ICC is used for response SEs and the predictor-specific ICC for predictor SEs. The response's ICC is never applied to predictor columns or vice versa, and it is the response's ICC (not the predictors') that sets `cell_weight` whenever a response was given. Requires at least 2 cells with 2+ observations and at least 2 residual

degrees of freedom (N - k >= 2); the ICC is taken as 0 — no correction, no "deff_applied" attribute — otherwise, and likewise when the estimate itself comes out at or below 0.

A positive number Applied as a uniform design effect to every cell, as sd * sqrt(deff / n) — exactly sqrt(deff) times the naive SE. Use when you have an external estimate of the design effect. Anything that is not a single number >= 1 — including a value below 1, which would *shrink* the standard errors — is refused with a warning and replaced by 1.

sac — Optional sac_range object from estimate_sac_range(), used when deff = "variogram". Supplying one avoids re-fitting the variogram and lets you inspect the fit the design effect is based on. A sac_range whose fit was *rejected* (its status is not "ok") carries no usable correlation function, so deff falls back to 1 with a warning rather than correcting by a shape that was not trusted enough to report a range.

deff_max_n — Cells with more than this many points are subsampled before forming the n x n correlation matrix used by deff = "variogram". Default 500.

quiet — Logical; suppress this function's progress message()s. It does not silence R warnings, nor the package's console log echo (see spatialkit_quiet for that). Default TRUE – unlike the tessellation functions, whose default is FALSE.

Details

This is the third step of the package's pipeline, taking the labelled layer from assign_features_to_polygons() down to cell level. What distinguishes it from a plain dplyr::group_by() + summarise() is that it carries the *uncertainty* of each aggregate with it: alongside every mean it returns a within-cell standard deviation, a standard error and an observation count, and it can correct that standard error for within-cell spatial autocorrelation via deff. Reach for it whenever the cell-level values will be modelled or mapped, because a cell mean over 2 observations and one over 200 are not the same measurement and nothing downstream can tell them apart otherwise.

In addition to user-specified aggregation functions, this function always computes within-cell standard deviation (..sd_<var>) and standard error (..se_<var>) for every numeric response/predictor column, plus an n column (rows falling in the cell) and a cell_weight column. These columns let downstream models account for the fact that a cell with 2 observations carries more aggregation uncertainty than one with 200.

cell_weight is the *effective* sample size of the primary variable — the response when one was supplied, otherwise the first predictor. It counts that variable's non-missing rows, not all rows (a cell of 10 rows with 3 finite responses carries 3 observations' worth of information about the response, not 10), and it is divided by that cell's design effect when deff applied one. With deff = 1 and no missing values it equals n. Pass it as the weights argument of a downstream regression.

Value

A tibble/data.frame (or sf if cells_sf given) with per-cell summaries: the ID column, n (rows in the cell — an input column also called n is not allowed to shadow it), one column per agg_funs entry per variable, ..sd_* / ..se_* for every numeric response and predictor, and cell_weight.

When a correction was actually applied, an attribute "deff_applied" is attached recording it: method plus icc_resp/icc_pred for "kish", deff/rbar/crs/max_n for "variogram", and deff

alone for a fixed number. When `cells_sf` is supplied, *every* per-cell vector in that attribute (`deff` and `rbar` alike) is realigned to the joined row order, so `deff[i]` and `rbar[i]` still describe row `i`; cells with no observations carry `NA`. No attribute is attached when no correction was applied — `deff = 1`, a `deff = "kish"` ICC of 0, or a `"variogram"` request that could not be fitted.

The ID column keeps its input type when `cells_sf`'s ID column and the summarised IDs already have the same class. When the classes differ, both are coerced to character in order to join (logged as a warning), and the returned ID column is therefore character.

Spatial autocorrelation and standard-error bias

Important: By default (`deff = 1`), the `..se_*` columns are computed as `sd / sqrt(n)`, which assumes observations within each cell are independent. When data are spatially autocorrelated — the common case for the spatial workflows this package supports — within-cell observations are typically positively correlated, so the effective sample size is smaller than `n`. The naive SE is therefore **anticonservative** (too small), and downstream weighted regressions using `cell_weight` or `..se_*` columns will produce overconfident standard errors for cells with strong intra-cell correlation.

Setting `deff = "kish"` applies an approximate correction using Kish's design effect. Separate intra-class correlations (ICCs) are estimated for response and predictor variables via a one-way random-effects decomposition across all cells. Each variable type's ICC is used for its own SE adjustment, and each cell's effective sample size is reduced to `n_i / (1 + (n_i - 1) * rho)`. This is a first-order correction that does not require a full spatial covariance model but does require enough cells and observations for a stable ICC estimate. You may also pass a fixed numeric design effect (e.g. `deff = 2`) to uniformly inflate standard errors: an externally supplied constant is applied as `sd * sqrt(deff / n)`, exactly `sqrt(deff)` times the naive SE in every cell. (The `E[s^2]` correction that the estimated design effects also apply is derived from within-cell correlation and would not be justified for a number the caller chose.)

Even with the Kish correction, the adjusted SE is an approximation. For rigorous inference under spatial dependence, consider fitting an explicit spatial covariance model (e.g. via `fit_bayesian_spatial_model`).

What the standard error estimates

The `..se_*` columns are the standard error of the cell mean **as an estimate of the population (grand) mean** — the unconditional quantity, in which the cell's own realised deviation is part of the error. That is the right quantity when cells are treated as samples from a common population, and the design-effect correction is calibrated for it: measured 95% interval coverage of the grand mean is 0.95 with `deff = "kish"` (and 0.29 with the naive SE) on exchangeable within-cell correlation, and 0.93 with `deff = "variogram"` on a simulated Gaussian field.

It is **not** the standard error of the cell's own mean (the block average over that cell), which is what a cell-level map or a regression on cell values usually wants. For that quantity the naive `sd / sqrt(n)` is the better of the two on offer — measured coverage 0.95 under exchangeable within-cell correlation, against essentially 1.00 for the design-effect-corrected SE, which is about five times too wide. That 0.95 is exact under the exchangeable model and holds under a spatial covariance model only when the cell's points are spread through the cell; with *clustered* sampling inside a cell it is anticonservative for the block average too (measured 0.58), and the honest answer there is a block-kriging variance, which this function does not compute. Use `deff` when the cell means feed a population-level inference; leave it at 1 when they are measurements of the cells themselves and the sampling within cells is reasonably uniform.

Design effects and variable types

`deff = "kish"` estimates a separate ICC for response and predictor variables and applies each to its own columns. `deff = "variogram"` fits or accepts **one** correlation function and applies it to every numeric column, because a variogram is a property of the field being modelled rather than of a variable type; a predictor whose spatial structure differs markedly from the response's will have its SE corrected by the response's correlation. The internally estimated variogram is fitted to the **response itself**, never to OLS residuals, whatever `predictor_vars` holds: the `..se_resp_*` columns estimate the grand mean of the response, so the correlation to correct for is the response's own. (A residual variogram, whose correlation is that of the part the predictors do not explain, is weaker; using it here dropped grand-mean coverage from 0.93 to 0.51 the moment a predictor was listed.) Pass `sac` explicitly when you want a different variogram – a residual one from `estimate_sac_range(..., predictor_vars = )`, say – and check `attr(sac, "detrended")` to know which you have.

See Also

`assign_features_to_polygons()`, which produces the input layer; `build_tessellation()` for the cells themselves.

Other aggregation: `assign_features_to_polygons()`, `determine_optimal_levels()`

Examples

```
library(sf)
set.seed(1)
n <- 200
x <- runif(n, 0, 100)
y <- runif(n, 0, 100)
# A response with spatial structure, so the within-cell ICC is not zero.
pts <- st_as_sf(
  data.frame(x = x, y = y, val = 0.05 * x + 0.05 * y + rnorm(n, sd = 0.5)),
  coords = c("x", "y"), crs = 32632
)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
grid <- create_grid_polygons(bnd, target_cells = 9, type = "square")
assigned <- assign_features_to_polygons(pts, grid)

# IID standard errors (default) vs Kish design-effect adjustment
naive <- summarize_by_cell(assigned, response_var = "val")
kish  <- summarize_by_cell(assigned, response_var = "val", deff = "kish")
data.frame(n = naive$n,
           se_naive = naive$..se_resp_val,
           se_kish  = kish$..se_resp_val)
attr(kish, "deff_applied")   # method, icc_resp, icc_pred, per-cell deff
```

`summary.spatial_fit` *Summarise a fitted spatial model*

Description

Computes goodness-of-fit metrics from `fitted(object)` against the observed response. A non-numeric response is an error – a character or factor response cannot be scored, and used to come back as `n = 0` with every metric `NA`; a logical response is treated as 0/1.

Usage

```
## S3 method for class 'spatial_fit'
summary(object, ...)
```

Arguments

`object`	A `spatial_fit` object.
`...`	Ignored.

Value

An object of class `summary.spatial_fit`: a list with `class`, `formula`, `n`, `response_var`, `predictor_vars`, `info` and `in_sample` (the metric data.frame, out-of-bag for an `rf_fit`).

What the metrics are computed on

For a `gwr_fit` or a `bayesian_fit` these are **in-sample** metrics: `fitted()` returns values computed at the training locations from the model that saw them. For an `rf_fit` they are **out-of-bag**, because `fitted.rf_fit()` returns out-of-bag predictions rather than in-sample ones (see `fit_rf_model`). The two are not comparable, and `print()` on the result labels which one it is holding, driven by `$info$fitted_are_oob`. Use `compare_models_cv` to compare backends.

Adjusted R-squared is suppressed: GWR's effective parameter count far exceeds the global predictor count, and a GP model has no simple `p`.

Percentage errors on responses with zeros

`MAPE` divides by the observed value and `SMAPE` by $|y| + |\hat{y}|$, so neither is defined where its denominator is zero. Rather than return `Inf` or `NaN`, both are averaged over the rows whose denominator is non-zero, and are `NA` when no row qualifies. **The returned value does not record how many rows that was**, and the `n` column counts finite observation/prediction pairs, not the rows either percentage error actually used.

This bites on any response taking exact zeros — counts, rainfall, abundance, claim amounts. On a zero-inflated response with 62 zeros out of 120, `MAPE` is an average over the 58 non-zero rows reported as though it covered all 120. `SMAPE` fails differently and more subtly: it drops the rows where observation and prediction are both near zero — which on a well-fitted zero-inflated model are the rows it got *right* — so it averages the harder rows only and reads worse than the fit deserves.

RMSE, MAE and R^2 use every finite row and are unaffected; prefer them whenever the response can be zero. For a Bayesian fit, cv_bayes() additionally reports CRPS and interval coverage, which are proper scoring rules and have no such failure mode.

---

voronoi_seeds_kmeans *K-means seed generation from point coordinates*

Description

Places k seed points at k-means cluster centres of the observed coordinates, so seeds — and the Voronoi cells built from them — follow the sampling density: clusters of observations attract seeds, empty ground gets none. Reach for this when you want cells that each carry a comparable number of observations, which is what makes per-cell aggregates in summarize_by_cell() similarly precise. Use voronoi_seeds_random() instead when you want coverage of the study area rather than of the data, and get_voronoi_seeds() to pick between them by name.

Usage

```
voronoi_seeds_kmeans(points_sf, k, set_seed = 456)
```

Arguments

points_sf	An sf object with POINT geometries.
k	Integer; requested number of clusters, and an upper bound rather than a guarantee. It is clamped, with a warning, to whichever is smaller of the number of distinct point positions and nrow(points_sf) - 1 — k-means can produce neither more centres than there are distinct points nor as many centres as there are rows. Check nrow() on the result.
set_seed	Optional integer RNG seed. Default 456.

Details

Lon/lat input is projected first so the k-means distances are metric rather than degrees. Rows with empty or non-finite coordinates are dropped with a warning, and k is clamped to the number of distinct positions.

Value

An sf object of **at most** k cluster-centre POINTs (fewer when k exceeds the number of distinct positions), with seed_id and method = "kmeans" columns matching get_voronoi_seeds().

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = runif(100, 0, 1000), y = runif(100, 0, 1000)),
  coords = c("x", "y"), crs = 32632
)
seeds <- voronoi_seeds_kmeans(pts, k = 8)
nrow(seeds)   # at most 8
```

---

voronoi_seeds_random *Random seed generation within a polygonal boundary*

Description

Draws k seed points uniformly at random inside boundary, ignoring where the observations are. Reach for this when the cells should cover the study area evenly — so that sparsely sampled ground still gets its own cells and is visibly under-sampled in the results — rather than concentrating resolution where the data already are, which is what voronoi_seeds_kmeans() does. It is also the honest choice for a null or sensitivity comparison: re-running an analysis over several random seedings shows how much of a result depends on one particular tessellation.

Usage

```
voronoi_seeds_random(boundary, k, set_seed = 456)
```

Arguments

boundary	An sf or sfc polygonal object.
k	Integer; number of random seeds.
set_seed	Integer RNG seed. Default 456.

Details

Sampling is by rejection inside the polygon, so an awkward geometry can return fewer than k seeds; that shortfall is warned about rather than silently padded.

Value

An sf object of **at most** k random POINTs (rejection sampling inside an awkward geometry can fall short of k, which is warned about), with seed_id and method = "random" columns matching get_voronoi_seeds().

Examples

```
library(sf)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(0, 0), c(100, 0), c(100, 100), c(0, 100), c(0, 0)
))), crs = 32632))
seeds <- voronoi_seeds_random(bnd, k = 10)
nrow(seeds)   # at most 10
```

Index