

Package: spatialkit (via r-universe)

October 7, 2026

Title Spatial Tessellation, Modeling, and Cross-Validation Toolkit

Version 2.0.0.9000

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://elkronos.github.io/gis_modeling_toolkit/,
https://github.com/elkronos/gis_modeling_toolkit

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

Encoding UTF-8

Roxygen list(markdown = TRUE)

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 (>= 3.4.0), patchwork, FNN, Matrix, nlme, spdep, testthat (>= 3.1.7), knitr, rmarkdown, units, vctrs, withr

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

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

Config/pak/sysreqs libabsl-dev cmake libgdal-dev gdal-bin libgeos-dev libssl-dev libproj-dev libsqlite3-dev libudunits2-dev

Repository <https://elkronos.r-universe.dev>

Date/Publication 2026-10-07 22:07:41 UTC

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

RemoteRef HEAD

RemoteSha 227c8e1ace732b69e27e40736b0b3dae51453e95

Contents

[.spatialkit_rows	4
area_of_applicability	5
assign_features_to_polygons	10
build_tessellation	12
clear_fitted_cache	15
clear_grid_cache	16
clip_target_for	17
coef.bayesian_fit	18
coef.gwr_fit	19
coef.rf_fit	20
coerce_to_points	21
compare_models	22
compare_models_cv	24
create_grid_polygons	28
create_grid_polygons_cached	30
create_voronoi_polygons	32
cv_bayes	34
cv_block_size_sweep	38
cv_gwr	40
cv_rf	43
cv_spatial	45
determine_optimal_levels	50
ensure_projected	55

ensure_stable_poly_id	58
estimate_sac_range	60
evaluate_insample	67
fit_bayesian_spatial_model	69
fit_gwr_model	75
fit_rf_model	78
fitted.bayesian_fit	81
fitted.gwr_fit	83
fitted.rf_fit	83
fold_separation	84
get_voronoi_seeds	86
gp_lengthscales_bounds	87
gwr_model_selection	89
harmonize_crs	92
kriging_adequacy	93
make_folds	97
model_metrics	105
new_spatial_fit	107
plot.aoa	109
plot.block_size_sweep	111
plot.feature_selection	112
plot.gwr_model_selection	113
plot.resolution_profile	114
plot.sac_range	115
plot.spatial_fit	116
plot_calibration	119
plot_cv_metrics	120
plot_folds	121
plot_tessellation_map	122
predict.bayesian_fit	125
predict.gwr_fit	126
predict.rf_fit	127
predict_surface	128
prep_model_data	131
print.aoa	133
print.gwr_model_selection	133
print.morans_i	134
print.resolution_profile	135
print.rf_fit	135
print.sac_range	136
print.spatial_fit	137
residual_morans_i	137
residuals.bayesian_fit	141
residuals.gwr_fit	142
residuals.rf_fit	143
resolution_profile	143
sac_nugget	148
select_features_forward	149

select_resolution	153
spatialkit_quiet	154
summarize_by_cell	155
summary.resolution_profile	163
summary.spatial_fit	164
voronoi_seeds_kmeans	166
voronoi_seeds_random	167

Index	169
--------------	------------

[.spatialkit_rows	<i>Subset a layer that carries a row record</i>
-------------------	---

Description

`prep_model_data()` and `assign_features_to_polygons()` return a layer with an attribute recording what happened to its rows ("dropped" and "ties" respectively). Those records describe the rows the layer was built with, and `[` on an `sf` object copies attributes through unchanged, which would leave a subset reporting its parent's numbers for a different set of rows. Subsetting therefore returns a plain layer with the record removed, and so do the **dplyr** verbs that select or reorder rows (`filter()`, `slice()`, `arrange()`, `distinct()`); read the record from the layer the function returned, before subsetting it. Binding such layers (`rbind()`, `dplyr::bind_rows()`) likewise returns a plain `sf` layer.

Usage

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

Arguments

<code>x</code>	A layer returned by <code>prep_model_data()</code> or <code>assign_features_to_polygons()</code> .
<code>...</code>	Passed to the underlying <code>sf</code> or data frame method.

Details

Each record carries `n_rows`, the number of rows it was made for. `sf::st_drop_geometry()` keeps the rows, and with them the record: it returns a data frame of class `c("spatialkit_rows", "data.frame")`. Binding such data frames with `rbind()` or `dplyr::bind_rows()` keeps the first one's record and class, so the record then describes only the first input's rows: its `n_rows` no longer equals `nrow()` of the result. The package's own readers ignore a record whose `n_rows` does not match; when reading `attr(x, "dropped")` or `attr(x, "ties")` yourself from a layer that has been through such steps, check it the same way.

Value

The subset, without the row records and without this class. A subset that is not a data frame (a single column taken with `drop = TRUE`) is returned unchanged.

See Also

[prep_model_data\(\)](#) and [assign_features_to_polygons\(\)](#), the two functions that attach the records this method removes.

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
)
clean <- prep_model_data(dat, "resp", "pred")
attr(clean, "dropped")$n      # 2
attr(clean[1:2, ], "dropped") # NULL -- the record does not follow
```

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 <code>include_coords = TRUE</code> an sf object is required, non-POINT geometry is reduced to representative points, and a CRS mismatch with the training data is reconciled; see <i>Models fitted with the coordinates as predictors</i> .
---------	---

model	A fitted <code>spatial_fit</code> , supplying the training data and predictor names. Optional if <code>train_sf</code> and <code>predictor_vars</code> 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 <code>include_coords = TRUE</code> , the coordinates <code>"..x"</code> and <code>"..y"</code> 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. Weights must be finite and non-negative, so pass permutation importance as <code>pmax(importance, 0)</code> . (A forest with no out-of-bag rows, <code>replace = FALSE</code> with <code>sample_fraction = 1</code> , has NaN importance, which <code>pmax()</code> keeps and which is refused; refit it with out-of-bag rows or use <code>weights = NULL</code> .) <code>pmax(importance, 0)</code> is all zero when the model found no predictor useful, and then the weights cannot say anything: with a single predictor any weight gives the same index and zero is accepted; with several, all of them are weighted equally, as with <code>weights = NULL</code> , and a warning says so. The coordinate default above is the mean of the supplied weights, so a zero weight on the only covariate of a coordinate-using model zeroes the coordinates too, and that equal weighting applies.
folds	Cross-validation folds: a <code>make_folds</code> result, a list of train/test splits, or a vector of fold labels with one entry per training row. Default <code>NULL</code> (plain nearest neighbour). The folds you passed to <code>cv_*</code> , built on the layer model was fitted from, may name rows that <code>prep_model_data()</code> removed (a missing or non-finite value, an empty geometry): as in <code>cv_*</code> , they are dropped from the folds, and a label vector with one entry per row of that layer loses theirs. Also as in <code>cv_*</code> , a <code>make_folds()</code> result built on other data – another layer, or these rows in another order – is refused. That check is skipped when the folds were built on polygons and the training data are the points a fit reduced them to.
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 <code>NULL</code> (chosen from the training size). Otherwise a single number of at least 1; a fractional value is truncated to a whole number of rows.
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, and a row outside the training range of a predictor in `dropped_vars` gets `DI = Inf` and `AOA = FALSE` (see *Limitations*).
- `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.
- `scaling`: a list with center and scale, each named by `predictor_vars`: the training means and standard deviations the index is computed in, so a location's DI can be traced to the predictor that put it outside.
- `n_outliers`: the number of training DI values above the $Q3 + 1.5 * IQR$ fence, which the default threshold rule sets aside (the "outlier-removed" in its name); computed whether or not threshold was supplied.
- `n_train`, `n_new`, `n_inside`, `n_outside`, `n_na`: row counts. `n_train` counts the training rows that survived the finite-value filter; `n_new` is every row of newdata, so `n_new = n_inside + n_outside + n_na`.
- `params` records the call: `folds_supplied`, `n_folds`, `folds_method`, `threshold_supplied`, `normalizer_max_n`, `normalizer_n_used`, `normalizer_subsampled`, `weights_supplied` and `seed`. `folds_method` is the method of a `make_folds()` result (`"block_kfold"`, `"random_kfold"`, ...), `"labels"` for a vector of fold labels, `"splits"` for a bare list of train/test splits, and NA when no folds were supplied. It is printed with the object, because the threshold's meaning depends on it: the cross-validated DI that sets it comes from the same kind of hold-out as the CV error it should be quoted beside, so an AOA built on `random_kfold` folds is logged as a caution and does not belong next to a blocked `cv_*` result.

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*. That means everything outside its own fold for random and block folds, and the smaller training set that buffered and NNDM folds actually leave

(see the next section). The threshold is then the largest training DI that is not an upper outlier, i.e. not above the fence $Q3 + 1.5 * IQR$ of the training DI, with the quartiles of `stats::quantile()`'s default type 7. Prediction points at or below that threshold are inside the AOA.

That is the paper's "outlier-removed maximum". **CAST**, the reference implementation, computes the same fence with the same quartiles but uses the fence itself as the threshold, capped at the largest training DI. The two agree whenever no training DI lies above the fence (`n_outliers` is 0); otherwise **CAST**'s threshold is the larger, and so is its AOA. (Earlier **CAST** releases used `grDevices::boxplot.stats()`, which gives the rule used here but with Tukey's hinges as the quartiles, so they can also differ when the number of training points is even.) To apply the current **CAST** rule to the same training DI, pass `threshold = min(quantile(res$train_DI, 0.75) + 1.5 * IQR(res$train_DI), max(res$train_DI))` for an earlier result `res`.

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.

Each training point's reference is its nearest *other* training row, so an exact duplicate in predictor space (repeat visits to a site with static covariates, or covariates read off a raster coarser than the sampling) has a training DI of 0. Once about three quarters of the rows have a twin among their reference rows the threshold is 0, and only exact copies of a training row count as inside. That is logged as a caution; folds that keep the duplicates together (`make_folds(method = "leave_location_out", group_var = ...)`), or removing them, give the threshold its meaning back.

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 and never 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 from the distance. A prediction point taking a value there outside the training range (with the same relative tolerance) is extrapolation along a direction the training data never varied in: its scaled distance along it is infinite, so it gets `DI = Inf`, is outside the AOA, and a warning gives the count. A point missing that value is judged on the other predictors. 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. That is 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. This is done 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](https://doi.org/10.1111/2041210X.13650)

See Also

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

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

Other prediction: `predict_surface()`

Examples

```
library(sf)
set.seed(1)
n <- 120
train <- st_as_sf(
  data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + 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 = 5e5 + runif(50, 0, 1000), y = 5e6 + 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$aaoa$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

<code>features_sf</code>	An sf object containing features to assign.
<code>polygons_sf</code>	An sf or sfc polygonal layer.
<code>polygon_id_col</code>	Name of the polygon identifier column. Default "poly_id".
<code>keep_unassigned</code>	Logical; retain features that fall inside no polygon, carrying NA in the ID column. Default FALSE, which drops them.
<code>predicate</code>	Binary spatial predicate function. Default <code>sf::st_intersects</code> . Not used when <code>largest</code> applies: sf then assigns polygon features by overlap area and never calls the predicate.
<code>largest</code>	Logical; when <code>features_sf</code> is itself polygonal, keep the polygon with the largest overlap. Default TRUE. Ignored for point and line features. A feature that only touches the polygon layer (shares an edge or a corner with it, with no overlap area) has no largest overlap and is unassigned, in any CRS; with <code>largest = FALSE</code> the default <code>st_intersects</code> counts touching, so such a feature is assigned. A feature that overlaps two or more polygons by exactly the same area (to 9 significant digits; a square split evenly across a cell edge) is given to one of them by <code>tie_break</code> and counted in "ties", so the choice does not depend on the order of the polygon rows. Invalid geometries (usually a self-intersecting ring), whose overlap is undefined, are repaired with <code>sf::st_make_valid()</code> for the join, with a warning, and returned as they arrived. If the overlap still cannot be computed the function stops: falling back to predicate and <code>tie_break</code> would change the rule for every feature in the layer, so pass <code>largest = FALSE</code> to ask for that.

tie_break	Strategy for resolving features that match multiple polygons (with largest, that overlap several polygons equally): "smallest_area" (default) keeps the polygon with the smallest area and, among polygons of equal area (a point on the shared edge of two grid cells), the one whose bounding-box centre is lowest, then leftmost, so the choice does not depend on the order of the rows; "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. Prefer it to `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.

The join runs in the CRS of `polygons_sf` whenever that CRS is projected, so cell edges are the straight lines the cells were drawn with and overlap areas are planar. A copy of `features_sf` is transformed for it, and the features come back with the coordinates they arrived with. Otherwise (the polygons are in lon/lat, or carry no CRS) the join runs in the CRS of `features_sf`.

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. A column of `features_sf` already called `polygon_id_col` is dropped before the spatial join (with a warning), so re-assigning an already-assigned layer replaces the old IDs and does not fail. Every other column is kept, including one named like the polygons' own ID column when that is read from a fallback such as "id" (a site id joined to cells keyed by id). 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). The attribute "ties" records how many features matched more than one polygon (with largest, overlapped several by exactly the same area) and had the `tie_break` rule decide for them: a list with `n`, which (their row positions in `features_sf`), `rule` and `n_rows` (the number of rows returned, which the record was made for). A large `n` means the polygon layer overlaps, and per-cell counts built from the result depend on the rule. The record describes the rows this call returned and does not survive subsetting: `joined[i, ]`, like `dplyr::filter()`, `slice()` or `arrange()` of it, is a plain layer with no "ties" attribute, so nothing reports the parent's count for a different set of rows. `sf::st_drop_geometry()` keeps the record, since the rows are the same; see `[.spatialkit_rows]` for what binding such data frames does.

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()`, `kriging_adequacy()`, `resolution_profile()`, `select_resolution()`, `summarize_by_cell()`, `summary.resolution_profile()`

Examples

```
library(sf)
```

```

set.seed(1)
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(20, 0, 100), y = 5e6 + runif(20, 0, 100),
            val = rnorm(20)),
  coords = c("x", "y"), crs = 32632
)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(5e5, 5e6), c(5e5 + 100, 5e6), c(5e5 + 100, 5e6 + 100),
  c(5e5, 5e6 + 100), c(5e5, 5e6)
))), 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	<i>Build a tessellation (Voronoi, Delaunay triangles, hex grid, or square grid)</i>
--------------------	---

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. Prefer it to 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. When exactly one of points_sf and boundary has a CRS, the other is interpreted in it, with a warning. CRS-less points, and a CRS-less boundary given with projected points, are read as [harmonize_crs\(\)](#) does; CRS-less points that do not look like lon/lat cannot take a geographic boundary's CRS and are refused with an error. A CRS-less boundary given with lon/lat points is read as lon/lat when its coordinates fit the lon/lat envelope, and refused with an error otherwise. When neither has one, both are read by the lon/lat heuristic of [ensure_projected\(\)](#): taken as EPSG:4326 and projected when the points look like degrees (a boundary whose coordinates do not fit the lon/lat envelope is then refused with an error), otherwise left in the same unnamed planar space.

method	One of "voronoi", "triangles", "hex", "square".
approx_n_cells	Approximate number of cells. Read by method = "hex" and "square" only: "voronoi" grows one cell per input point and "triangles" one triangle per neighbouring triple, so neither has a count to set, and both warn that the argument was ignored. For a Voronoi cell count, place the seeds with get_voronoi_seeds() and tessellate those. For hex grids the target is adjusted for packing density; the actual count after clipping to an irregular boundary may differ noticeably. Besides a number, this accepts what the level-selection step returned: the integer vector of ranked candidates from determine_optimal_levels() (its first element is used), a select_resolution() result (its \$best), or a resolution_profile() (read with select_resolution() at its default criterion). The count used is returned as params\$approx_n_cells and where it came from as params\$approx_n_cells_from (NULL for a plain number). A count read off a profile or selection is a number of k-means cells: every one occupied, and small where the points are dense. A lattice lays that many equal cells over the whole boundary, so on clustered points many of them hold no point (about half, on six clusters in a square); on evenly spread points it matches. params\$cells_occupied and params\$cells_empty report how the points filled the grid, and a count that came from a profile or selection warns when fewer than three quarters of it are occupied. For cells that follow the points, seed a Voronoi tessellation with get_voronoi_seeds(method = "kmeans", n = <the selection>, sample_points = <the points>) : given the select_resolution() result or the profile itself, it returns the centres of the partition the profile scored, whose Voronoi cells are the cells the criteria judged.
cellsize	Numeric cell size, in the units of the working CRS. Read by method = "hex" and "square" only; the other two methods warn that it was ignored. When both cellsize and approx_n_cells are given, cellsize wins and approx_n_cells is ignored with a logged warning; supply one or the other. With a geographic crs it is in that CRS's degrees, and the grid is laid in degrees.
expand	Buffer distance, in the working CRS's units, by which the Voronoi boundary is grown before the diagram is built. Applied by method = "voronoi" only; the "hex", "square" and "triangles" methods ignore it (the value you passed is still echoed back in params\$expand). With clip = TRUE the cells are clipped to the grown boundary, which is the one returned as boundary: cells reach expand beyond the study area, and a point up to expand outside it is indexed.

<code>clip</code>	Logical; clip to boundary.
<code>keep_duplicates</code>	Logical. Has no effect on the cells or the index: coincident points are merged before a Voronoi diagram or a Delaunay triangulation is built either way, and every one of them is indexed to the cell they share.
<code>crs</code>	Optional target CRS: anything <code>sf::st_crs()</code> accepts, including an sf or sfc layer, whose CRS is used. A projected CRS is the working CRS. A geographic one (EPSG:4326, say) is the CRS the result is returned in: the cells are built in the local projected CRS <code>ensure_projected()</code> picks for the points, indexed there, and then transformed with long edges densified, so Voronoi cells are nearest-point cells on the ground and grid cells are laid in metres rather than degrees. The exception is a hex or square grid sized by <code>cellsize</code> , which is in degrees and so is laid in degrees. Whenever that local CRS is picked for lon/lat points, or CRS-less ones taken as lon/lat (no <code>crs</code> , or a geographic one), a hex or square grid with a boundary is laid in it unless it distorts areas across the boundary by more than 1 percent (Web Mercator over a near-global extent, say); the grid is then laid, and the points indexed, in the equal-area CRS <code>ensure_projected(purpose = "area")</code> picks for the boundary, with a logged warning, as <code>create_grid_polygons()</code> does, so the cells stay equal-area.
<code>quiet</code>	Logical; suppress this function's progress message(s). It does not silence R warnings, nor the package's console log echo (see <code>spatialkit_quiet</code> for that). Default FALSE.

Details

Which method to reach for. "voronoi" gives one cell per point, so resolution follows sampling density. That is 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, and the data does not make it 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; it is not meant as an aggregation unit. `determine_optimal_levels()` will suggest a cell count from the spatial structure of the data.

"voronoi" and "triangles" are built on the points' vertices: a MULTIPOINT feature with several vertices gets one cell (or triangle corner) per vertex, and its index entry is the smallest `cell_id` among the cells it touches. See `create_voronoi_polygons()`.

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, that is, one outside the study area. Only a point within a thousandth of the median cell width of a cell is snapped to it. That covers points sitting exactly on a shared edge, and leaves points outside the study area as NA. A summary built from `index` therefore counts only the points the tessellation actually covers.

boundary The boundary used (possibly derived and/or reprojected, and for "voronoi" grown by expand).

method The method actually used.

params The parameters the tessellation was built with, plus snapped, the record of that nearest-cell repair: a list with n, which (row positions in points_sf) and distance (how far outside every cell each sat, in CRS units). For "hex" and "square" also cells_occupied and cells_empty, the number of cells that hold at least one point and that hold none.

See Also

Other tessellation: [create_grid_polygons\(\)](#), [create_grid_polygons_cached\(\)](#), [create_voronoi_polygons\(\)](#), [ensure_stable_poly_id\(\)](#), [get_voronoi_seeds\(\)](#), [plot_tessellation_map\(\)](#), [voronoi_seeds_kmeans\(\)](#), [voronoi_seeds_random\(\)](#)

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(20, 0, 100), y = 5e6 + runif(20, 0, 100)),
  coords = c("x", "y"), crs = 32632
)
tess <- build_tessellation(pts, method = "voronoi", quiet = TRUE)
tess$cells
```

clear_fitted_cache	<i>Clear cached fitted values for a Bayesian spatial model</i>
--------------------	--

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).

See Also

Other package options and caches: [clear_grid_cache\(\)](#), [create_grid_polygons_cached\(\)](#), [spatialkit_quiet\(\)](#)

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 = 5e5 + runif(60, 0, 1000), y = 5e6 + 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	<i>Clear the in-session grid cache</i>
------------------	--

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.

See Also

Other package options and caches: [clear_fitted_cache\(\)](#), [create_grid_polygons_cached\(\)](#), [spatialkit_quiet\(\)](#)

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() # returns the number of entries removed, invisibly
print(clear_grid_cache()) # 0: already empty
```

clip_target_for	<i>Build a polygonal clip target from points and/or a boundary</i>
-----------------	--

Description

Resolves a single polygon to tessellate within. With a boundary it is that boundary (optionally buffered by `expand`); without one it is the axis-aligned bounding box of `points_sf` (the rectangle in the working CRS), again optionally buffered, or a small buffer around the points when they all share one `x` or one `y`, or nearly so (the short side of their bounding box below a millionth of the long side). Reach for it to build the boundary that `method = "hex"` and `"square"` require, 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

<code>points_sf</code>	An <code>sf</code> object with POINT/MULTIPOINT geometry.
<code>boundary</code>	Optional polygonal <code>sf</code> object. One with no CRS, given with points that have one, is interpreted in the points' own CRS, with a warning. With points in a projected CRS it is read as <code>harmonize_crs()</code> does: coordinates that look like lon/lat are taken as EPSG:4326 and reprojected, anything else is stamped with the points' CRS. With lon/lat points it is read as lon/lat when its coordinates fit the lon/lat envelope, and refused with an error otherwise.
<code>expand</code>	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 <code>sf::st_buffer()</code> interprets <code>dist</code> 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 <code>@return</code>), so both forms agree.
<code>quiet</code>	Logical; suppress this function's progress message()s. It does not silence R warnings, nor the package's console log echo (see <code>spatialkit_quiet</code> for that). Default FALSE.

Details

It is not the target `build_tessellation()` derives on its own: `method = "voronoi"` without a boundary clips to the convex hull of the points buffered by 2 percent of its diagonal, and there expand is always a distance. A bounding box over a non-rectangular point cloud includes corners with no data, so a hex or square grid laid over it has cells that hold no points; pass the study-area polygon when there is one.

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`.

See Also

Other spatial data preparation: `coerce_to_points()`, `ensure_projected()`, `harmonize_crs()`, `prep_model_data()`

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(30, 0, 100), y = 5e6 + runif(30, 0, 100)),
  coords = c("x", "y"), crs = 32632
)
# No boundary: the bounding box, expanded by 10% of the extent
box <- clip_target_for(pts, expand = 0.1, quiet = TRUE)
st_area(box)
```

`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()`, on the fitted scale (per standard deviation of each predictor under `standardize_predictors = TRUE`; see above). Never NULL: a missing 'brms' or a failing `fixef()` call errors, following the `coef()` contract described in [new_spatial_fit](#).

Standardised predictors

The summaries are on the scale the model was fitted on. A fit made with `standardize_predictors = TRUE` was fitted on centred and scaled numeric predictors, so each slope is the change in the linear predictor per *standard deviation* of its predictor and the intercept is its value at the predictor *means*, not the raw-unit numbers `stats::lm()` reports on the same formula. Nothing on the returned matrix says so; `print()` on the fit does, and the centre and scale of each predictor are in `object$info$predictor_scaling`. To put a slope back in raw units divide its Estimate, Est.Error and interval bounds by that predictor's scale. The intercept's Estimate follows by linearity (subtract each raw-unit slope times its predictor's center), but its Est.Error and interval depend on the posterior covariance of the coefficients: transform the draws from `brms::as_draws_df(object$engine)` for those, or refit without standardising.

See Also

Other methods on a fitted model: [coef.gwr_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.gwr_fit\(\)](#), [fitted.rf_fit\(\)](#), [model_metrics\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [print.spatial_fit\(\)](#), [residuals.bayesian_fit\(\)](#), [residuals.gwr_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

coef.gwr_fit

*Extract GWR local coefficients***Description**

Returns the whole surface of coefficients, one row per observation and one column per term, against 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. The call neither errors nor returns a silently short table.

See Also

Other methods on a fitted model: [coef.bayesian_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.gwr_fit\(\)](#), [fitted.rf_fit\(\)](#), [model_metrics\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [print.spatial_fit\(\)](#), [residuals.bayesian_fit\(\)](#), [residuals.gwr_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

coef.rf_fit

Coefficients are undefined for a random forest

Description

Consistent with `coef.gwr_fit()` and `coef.bayesian_fit()`, which also error, instead of 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.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

code 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

<code>x</code>	An sf object.
<code>mode</code>	One of "auto", "centroid", "point_on_surface", "surface", "line_midpoint", "bbox_center".
<code>tmp_project</code>	Logical; temporarily project for line-based midpoints. When <code>x</code> has no CRS and the lon/lat heuristic of <code>ensure_projected()</code> takes its coordinates for degrees (inside the lon/lat envelope and more than one unit across, or with decimal-degree precision), 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 <code>tmp_project = FALSE</code> , for planar data.

Details

The result has one row per row of `x`, in the same order. An EMPTY geometry of any type, lines included, becomes an EMPTY POINT in its own row; `prep_model_data()` and `make_folds()` then drop such rows, as they drop any other empty geometry. Empty lines are never handed to `sf::st_line_sample()`: it yields no midpoint for them, which would misalign the result, and with sf 1.0.x an empty MULTILINESTRING (or an empty part of one) crashed the R session. An empty part inside a non-empty feature is ignored, so the feature gets the point its other parts give; GEOS's interior point, used by "point_on_surface" and by the temporary projection's choice of CRS, segfaulted on an empty line part too.

Value

An sf object with geometry coerced to POINTs, row for row with x; an empty input geometry gives an empty POINT.

See Also

Other spatial data preparation: [clip_target_for\(\)](#), [ensure_projected\(\)](#), [harmonize_crs\(\)](#), [prep_model_data\(\)](#)

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 (for a forest, out-of-bag) metrics and model-specific information criteria (AICc, LOOIC).

Usage

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

Arguments

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

Value

A data.frame comparing all models. Its `metric_basis` column says what each row's metrics were computed on (see [evaluate_insample](#)); a table that mixes "out-of-bag" and "in-sample" rows does not rank the models, and says so in the log. AICc (GWR) and LOOIC (Bayesian) are sums over the rows a model was fitted to, so each column is set to NA, with a warning, when the fits carrying it were fitted to different rows (a predictor with missing values drops rows, for example) or to different responses (a transformed response on the same rows), and the warning says

which. `convergence_ok` is TRUE or FALSE for a Bayesian fit whose convergence was checked (see [fit_bayesian_spatial_model](#)) and NA otherwise; a fit that did not converge is ranked like the others, so it also raises a warning. Alongside the metrics it carries `resid_morans_I`, `resid_morans_z`, `resid_morans_p` and `resid_morans_null`, the last of which names the null [residual_morans_i](#) scored each model against, since that choice is per-fit and governs how much the p-value is worth. A significant p-value is noted in the log (not raised as an R warning): positive autocorrelation as structure the model may have missed, negative (`resid_morans_z < 0`) as the alternating residuals of a model that tracks its data closely. Read the caveats in `?residual_morans_i` before treating silence as evidence of no residual structure.

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 data.frame `model_metrics()` returns carries no label distinguishing the two, so check `object$info$fitted_are_oob` before comparing numbers across backends; [evaluate_insample\(\)](#) and [compare_models\(\)](#) record it per model in a `metric_basis` column. [compare_models_cv](#) scores every backend the same way.

R^2 is $1 - RSS/TSS$ with the total sum of squares taken about the mean of the response the model was *fitted* to. In sample that is the ordinary R^2 . With `newdata` it is out-of-sample R^2 , the convention every `cv_*`() function uses: the model is measured against the prediction it had to beat, the training mean, not against the new rows' own mean, which it could not have known. It is below 0 when the model predicts the new rows worse than the training mean does, and it is NA when the response does not vary about that baseline by more than rounding error (100 machine epsilons of its magnitude, whatever its units).

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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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\(\)](#), [model_metrics\(\)](#), [residual_morans_i\(\)](#)

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = 5e5 + runif(60, 0, 1000), y = 5e6 + runif(60, 0, 1000),
              a = rnorm(60), b = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  # An offset keeps the response away from zero, where MAPE is undefined.
  pts$z <- 10 + 2 * pts$a - pts$b + rnorm(60, 0, 0.3)
  # Two predictor sets for the same learner: the model that leaves `b` out
  # against the one that has it. These are in-sample (out-of-bag) numbers;
  # compare_models_cv() scores the same question on spatial folds.
  fits <- list(a_only = fit_rf_model(pts, "z", "a", num_trees = 100, seed = 1),
              a_and_b = fit_rf_model(pts, "z", c("a", "b"), num_trees = 100, 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,
  pointsize = "auto",
  gwr_args = list(),
  bayes_args = list(),
  rf_args = list(),
  summary = c("mean", "median"),
  quiet = FALSE,
  block_size = NULL,
  auto_range = FALSE,
  metrics = NULL
)
```

Arguments

data_sf	An sf object.
response_var	Response column name.
predictor_vars	Predictor column names.
models	Character vector: any subset of <code>c("GWR", "Bayesian", "RF")</code> , 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. There is no 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 <i>none</i> of the requested backends is installed, nothing is left to compare and the call errors with "no viable models.". Guard with <code>requireNamespace()</code> 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 <code>make_folds()</code> return value, or a bare list of <code>list(train =, test =)</code> pairs of <code>..row_id</code> 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. It also decides where a polygon or line row falls in the shared blocks, so each row is placed by the point every model is fitted at.
gwr_args	Extra arguments for <code>cv_gwr</code> . Only names that are formal arguments of <code>cv_gwr()</code> are forwarded (it has no <code>...</code>), so entries meant for <code>fit_gwr_model()</code> alone (e.g. <code>longlat</code>) cannot be passed this way. Anything dropped is named in a warning; call <code>cv_gwr()</code> directly if you need it.
bayes_args	Extra arguments for <code>fit_bayesian_spatial_model()</code> . Forwarded whole as <code>cv_bayes(fit_args =)</code> , so an unrecognised name raises an error from <code>fit_bayesian_spatial_model()</code> and is not dropped silently. <code>compute_loo</code> , <code>boundary</code> and <code>pointize</code> are overridden by the CV internals.
rf_args	Extra arguments for <code>cv_rf</code> , which passes anything it does not recognise on to <code>fit_rf_model</code> and thence to <code>ranger::ranger()</code> .
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 <code>spatialkit_quiet</code> for that). Default FALSE.
block_size	Optional minimum block edge length for the shared spatial CV blocks (projected CRS units), passed to <code>make_folds()</code> when <code>folds</code> is NULL. Default NULL.
auto_range	Logical. If TRUE and <code>folds</code> is NULL, the autocorrelation range of the response (detrended on <code>predictor_vars</code>) is estimated and used as the minimum block size of the shared folds, as in <code>make_folds()</code> . Default FALSE: geometric blocks, as before this argument existed. Either way the fold set is built once and every

backend is scored on it. When it cannot be built (a `block_size` or estimated range that leaves a single block, say) the call is an error, as it is for each backend on its own; no model is scored on a design other than the one asked for.

metrics Optional scoring function of your own, handed to every backend's `cv_*`(): a function(`y`, `yhat`) returning a named numeric vector, applied per fold and to each backend's pooled predictions, whose names become columns of `by_fold` and `overall` beside the built-in ones. See **Your own metrics** on `cv_spatial()` for the contract. Because the backends are scored on the same folds and, in `overall`, on the same rows (see **Value**), the columns are comparable across rows of `overall`.

Value

A list with `overall`, `by_fold`, and per-model `cv_results` (`gwr_cv`, `bayes_cv`, `rf_cv` for the models that ran). `overall` has one row per model with the pooled metrics, `n_pred` (the rows they are computed on), the coverage and CRPS columns described above when a Bayesian model ran, and `model` as its last column. Shared folds do not guarantee shared rows: a model that fails on a fold (GWR with a fixed bandwidth across a gap in the data, say) or predicts NA for some rows pools fewer rows, usually without the hardest ones. When the models predicted different rows, the function warns and recomputes every model's pooled metrics, your own `metrics` included, on the rows all of them predicted, so `n_pred` is the same on every row that has predictions; a model that predicted nothing stays an NA row. Each model's metrics over all the rows it predicted stay in its `*_cv` element and in `attr(overall, "all_rows")`, a table of the same shape. `by_fold` and the Bayesian coverage and CRPS columns are per fold and are not recomputed. Only the models that actually ran appear, so check which names are present; there is not always one entry per requested model, because 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.

Coverage and CRPS in the overall table

A model can predict well on average and still be badly calibrated, so a comparison read from RMSE alone can prefer the model whose uncertainty is wrong. Heaton et al. (2019) found good point prediction routinely alongside poor interval coverage. When "Bayesian" is among the models that ran, `overall` therefore also carries the columns of `cv_bayes()`\$`predictive_coverage`: `coverage_50`, `coverage_80`, `coverage_95` (the share of held-out rows inside the posterior predictive interval at each level, averaged across folds weighted by each fold's `n_pred`) and `mean_CRPS` (the continuous ranked probability score, lower is better). They are NA on the GWR and RF rows, which produce point predictions and no draws, and absent when no Bayesian model ran. Read coverage against its nominal level: 0.95 at `coverage_95` is calibrated, well below it is overconfident, well above it is wider than it needs to be.

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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows

that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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.

Which metrics survive a non-Gaussian response

RMSE and MAE are defined for any numeric response and are what to read for a count, a rate or a bounded outcome. MAPE and SMAPE assume a response that is rarely zero (see the previous section), and R-squared and adjusted R-squared compare residual variance to total variance, which is the right comparison for a Gaussian response and a loose one for anything whose variance tracks its mean. None of the four is wrong to compute; each is Gaussian-shaped thinking, and on a Poisson or zero-inflated response should be read as a rough summary rather than a score.

For the Bayesian backend, `cv_bayes()` additionally reports CRPS and interval coverage at 50, 80 and 95 percent. Both are proper scoring rules computed from posterior draws, so they are meaningful for any family that predicts one number per row (a count, a rate, a binary or bounded outcome), and they are the numbers to compare when the response is not Gaussian. A categorical or ordinal family predicts a probability per response category instead, so `cv_bayes()` refuses one before fitting anything. When every fold fails, the `fold_metrics` frame `cv_bayes()` returns carries the CRPS column but not the `coverage_*` columns, so code that reads those columns must tolerate their absence.

References

Heaton, M. J., Datta, A., Finley, A. O., Furrer, R., Guinness, J., Guhaniyogi, R., Gerber, F., Gramacy, R. B., Hammerling, D., Katzfuss, M., Lindgren, F., Nychka, D. W., Sun, F. and Zammit-Mangion, A. (2019). A case study competition among methods for analyzing large spatial data. *Journal of Agricultural, Biological and Environmental Statistics*, 24(3), 398–425. doi:10.1007/s1325301800348w

See Also

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

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  dat <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
```

```

        elev = rnorm(n)),
        coords = c("x", "y"), crs = 32632
    )
    dat$price <- 10 + 0.01 * (st_coordinates(dat)[, 1] - 5e5) +
      2 * dat$elev + rnorm(n)
    # The random forest, and GWR beside it when GWmodel is installed, on the
    # same folds. "Bayesian" is left out here because a Stan fit per fold
    # takes minutes; add it to `models` for a run to report.
    models <- c("RF", if (requireNamespace("GWmodel", quietly = TRUE) &&
      requireNamespace("sp", quietly = TRUE)) "GWR")
    cmp <- compare_models_cv(dat, "price", "elev", models = models, k = 3,
      rf_args = list(num_trees = 100),
      gwr_args = list(bandwidth = 30))
    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, instead of one dictated by where the observations happen to be. That is the case when you need per-cell rates comparable across the map, or a resolution that stays fixed as the sample grows. 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

<code>boundary</code>	Polygonal sf or sfc object.
<code>target_cells</code>	Optional approximate desired number of cells. The cell <i>size</i> is derived from it as $\sqrt{\text{area} / \text{target_cells}}$, where area is that of the boundary's bounding box, so square grids get square cells; for hex grids the count is adjusted

for hexagonal packing density and the size rounded so that a whole number of hexagon widths spans the longer side of the box. The size does not depend on which way the boundary lies; the count can, because hexagon rows are 0.87 cellsize apart while columns are cellsize apart (about 10 percent on a moderately elongated box, up to 1.7 times on a strip narrower than one hexagon). 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. <code>sf::st_make_grid()</code> derives hexagon placement from cellsize alone, so for type = "hex" n does not set the number of cells, although 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: anything <code>sf::st_crs()</code> accepts, including an sf or sfc layer, whose CRS is used. When NULL (default) a lon/lat boundary is projected with <code>ensure_projected()</code> , which changes the CRS of the returned grid; a message reports this unless quiet = TRUE. When that CRS would distort cell areas across the boundary by more than 1 percent (Web Mercator over a near-global extent, a UTM zone stretched well past its width), <code>ensure_projected(purpose = "area")</code> is used instead, with a logged warning, so the cells stay equal-area; a local extent keeps its UTM zone. A geographic crs (EPSG:4326, say) is the CRS the grid is returned in: a grid sized by target_cells or n is laid in that projected CRS and then transformed, with long edges densified. A cellsize is in the units of crs, so with a geographic crs it is in degrees and the grid is laid in degrees, as asked; such cells are not equal-area.
quiet	Logical; suppress this function's progress message(s). It does not silence R warnings, nor the package's console log echo (see <code>spatialkit-quiet</code> 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 before memory is exhausted. 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_grid_polygons_cached\(\)](#), [create_voronoi_polygons\(\)](#), [ensure_stable_poly_id\(\)](#), [get_voronoi_seeds\(\)](#), [plot_tessellation_map\(\)](#), [voronoi_seeds_kmeans\(\)](#), [voronoi_seeds_random\(\)](#)

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

Arguments

<code>boundary</code>	An sf or sfc polygonal object.
<code>target_cells</code>	Approximate desired number of cells. Default NULL, as in create_grid_polygons() , so the grid can be sized by <code>cellsize</code> or <code>n</code> passed through <code>...</code> instead.
<code>type</code>	Grid type: "square" (the default) or "hex", matching create_grid_polygons() .
<code>...</code>	Additional arguments forwarded to create_grid_polygons() .
<code>cache_env</code>	Environment for memoized grids. Default <code>.gmt_cache</code> .
<code>max_entries</code>	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. 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.

See Also

Other tessellation: [build_tessellation\(\)](#), [create_grid_polygons\(\)](#), [create_voronoi_polygons\(\)](#), [ensure_stable_poly_id\(\)](#), [get_voronoi_seeds\(\)](#), [plot_tessellation_map\(\)](#), [voronoi_seeds_kmeans\(\)](#), [voronoi_seeds_random\(\)](#)

Other package options and caches: [clear_fitted_cache\(\)](#), [clear_grid_cache\(\)](#), [spatialkit_quiet\(\)](#)

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)
# The IDs come from ensure_stable_poly_id(), so they follow the geometry:
# the same request with the boundary's vertices in another order gives the
# same ID to the same cell.
bnd2 <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(100, 100), c(0, 100), c(0, 0), c(100, 0), c(100, 100)
))), crs = 32632))
g2 <- create_grid_polygons_cached(bnd2, target_cells = 16, type = "hex")
same_cell <- match(st_as_text(st_geometry(g2)), st_as_text(st_geometry(g)))
all(g2$poly_id == g$poly_id[same_cell])
```

```
create_voronoi_polygons
```

Create Voronoi polygons from points with CRS handling 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. Points with no CRS whose coordinates look like lon/lat (the heuristic ensure_projected() applies, with its warning) are taken as EPSG:4326 and projected, as lon/lat points are.
boundary	Optional polygonal sf object. When exactly one of points_sf and boundary has a CRS, the other is interpreted in it, with a warning. CRS-less points, and a CRS-less boundary given with projected points, are read as harmonize_crs() does (lon/lat-looking coordinates are reprojected from EPSG:4326, others are stamped); CRS-less points that do not look like lon/lat cannot take a geographic boundary's CRS, and are refused with an error. A CRS-less boundary given with lon/lat points (or with CRS-less points taken as lon/lat) is read as lon/lat when its coordinates fit the lon/lat envelope, and refused with an error otherwise.
expand	Numeric; absolute distance, in the working CRS's units, by which the boundary (or the hull derived from the points) is grown before the diagram is built. With clip = TRUE the cells are clipped to the grown boundary, so they reach expand beyond the study area, and a point up to expand outside it gets a cell and an index value. The grown boundary is the one returned as boundary.
clip	Logical; intersect cells with boundary.

keep_duplicates	Logical. Has no effect on the result: coincident points are merged before the diagram is built either way, so they share one cell and all of them are indexed to it.
crs	Optional target CRS: anything <code>sf::st_crs()</code> accepts, including an sf or sfc layer, whose CRS is used. A projected CRS is the working CRS. A geographic one (EPSG:4326, say) is the CRS the result is returned in: the cells are built in the local projected CRS <code>ensure_projected()</code> picks for the points, so they are nearest-point cells on the ground, and are then transformed, with long edges densified.
quiet	Logical; suppress this function's progress message(s). It does not silence R warnings, nor the package's console log echo (see <code>spatialkit_quiet</code> 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.

The generators are the points' vertices, not the features. A MULTIPOINT feature with several vertices therefore gets one cell per vertex, and its `index` entry is the smallest `cell_id` among the cells it touches; the others are referenced by no feature. Other functions in the package (`prep_model_data()`, `make_folds()`) reduce such a feature to its centroid instead, so cast to POINT, or take centroids, first if one cell per feature is what you want.

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, which means outside the study area (grown by `expand` when it is positive), so a summary built from it counts only the points the tessellation actually covers. `boundary` is the boundary the cells were built in: the one supplied or derived, grown by `expand`.

See Also

Other tessellation: `build_tessellation()`, `create_grid_polygons()`, `create_grid_polygons_cached()`, `ensure_stable_poly_id()`, `get_voronoi_seeds()`, `plot_tessellation_map()`, `voronoi_seeds_kmeans()`, `voronoi_seeds_random()`

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(15, 0, 100), y = 5e6 + 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% intervals, and `mean_CRPS` rates sharpness and 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,
  pointsize = "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,
  metrics = NULL
)
```

Arguments

<code>data_sf</code>	An sf object.
<code>response_var</code>	Response column name.
<code>predictor_vars</code>	Predictor column names.
<code>folds</code>	Optional fold definitions, in any of three shapes: a <code>make_folds()</code> return value; a bare list of <code>list(train =, test =)</code> pairs of <code>..row_id</code> 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 (<code>blockCV::cv_spatial()</code>)

	returns one as <code>\$folds_ids</code>), since its <code>\$folds_list</code> holds two <i>unnamed</i> 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. 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 <code>fit_args</code> 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 <code>fit_bayesian_spatial_model()</code> . A user-supplied <code>gp_k</code> is respected in every fold; when omitted, the GP rank is auto-selected per training fold. <code>compute_loo</code> , <code>boundary</code> , and <code>pointize</code> are always overridden by the CV internals. A categorical or ordinal family (<code>brms::categorical()</code> , <code>cumulative</code> , <code>sratio</code> , <code>cratio</code> , <code>acat</code>) is refused before anything is fitted: its prediction is a probability per response category, and every score here needs one number per row.
summary	"mean" or "median" for posterior predictions.
compute_pred_intervals	Logical; compute predictive intervals.
coverage_levels	Numeric vector of the nominal coverage levels to score, as proportions strictly between 0 and 1 (0.95, not 95), each given once; anything else is an error. Each becomes a column <code>coverage_<percent></code> of <code>fold_metrics</code> , named at full precision (0.975 gives <code>coverage_97.5</code>).
block_size	Optional minimum block edge length for spatial CV blocks (projected CRS units).
auto_range	Logical. If TRUE and <code>folds</code> 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 <code>parallel::mclapply()</code> (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. Mind the memory: every fold compiles its own Stan model, and one compilation can take several GB (3.6 GB was measured), so <code>parallel = n</code> runs <code>n</code> of them at once. A compiler killed for lack of memory fails its fold with rstan's "invalid connection" error, which <code>fold_status</code> records; use fewer cores if you see it.
metrics	Optional scoring function of your own, a function(<code>y</code> , <code>yhat</code>) returning a named numeric vector; its names become columns of <code>fold_metrics</code> (per fold) and overall (pooled) beside the built-in ones. See Your own metrics on <code>cv_spatial()</code> for the contract. <code>yhat</code> is the posterior predictive summary (mean or median) of each held-out row; a score that needs the draws belongs in <code>predictive_coverage</code> and CRPS, which this function computes itself.

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`, `fold_status`, `orphan_rows`, `n_unknown_ids`, `n_dropped`, `formula`, `predictive_coverage` and `coverage_levels` (the nominal levels, named by their `coverage_*` column). The two fold counts make a run where every fold failed visible in the return value itself, beyond the warning, and `fold_status` (one row per fold: `fold`, `status`, `message`) keeps the reason each missing fold is missing, including the error text of a fold whose sampler failed, where a long run's console output would not. See `cv_spatial` for the five statuses and for `orphan_rows`. `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`. `fold_metrics` carries, beyond the columns its siblings share, `gp_k`, `gp_n_basis`, `n_draws`, CRPS, the `coverage_*` columns and `convergence_ok`: TRUE or FALSE as `fit_bayesian_spatial_model()` judged that fold's sampler (R-hat, effective sample size, divergences), NA when `fit_args` sets `check_convergence = FALSE`. A fold that did not converge is scored like the others, so a run with any FALSE raises one warning naming those folds. 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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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.

Which metrics survive a non-Gaussian response

RMSE and MAE are defined for any numeric response and are what to read for a count, a rate or a bounded outcome. MAPE and SMAPE assume a response that is rarely zero (see the previous section), and R-squared and adjusted R-squared compare residual variance to total variance, which is the right comparison for a Gaussian response and a loose one for anything whose variance tracks its mean. None of the four is wrong to compute; each is Gaussian-shaped thinking, and on a Poisson or zero-inflated response should be read as a rough summary rather than a score.

For the Bayesian backend, `cv_bayes()` additionally reports CRPS and interval coverage at 50, 80 and 95 percent. Both are proper scoring rules computed from posterior draws, so they are meaningful for any family that predicts one number per row (a count, a rate, a binary or bounded outcome), and they are the numbers to compare when the response is not Gaussian. A categorical or ordinal family predicts a probability per response category instead, so `cv_bayes()` refuses one before fitting anything. When every fold fails, the `fold_metrics` frame `cv_bayes()` returns carries the CRPS column but not the `coverage_*` columns, so code that reads those columns must tolerate their absence.

See Also

Other cross-validation: `area_of_applicability()`, `cv_block_size_sweep()`, `cv_gwr()`, `cv_rf()`, `cv_spatial()`, `estimate_sac_range()`, `fold_separation()`, `gwr_model_selection()`, `make_folds()`, `sac_nugget()`, `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 = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * (st_coordinates(dat)[, 1] - 5e5) +
    2 * dat$elev + rnorm(n)
  # Two short chains per fold keep this to a few minutes and leave the
  # posterior rough, so the intervals below will run narrow; a run to
  # report uses brms's defaults (chains = 4, iter = 2000).
  cv <- cv_bayes(dat, "price", "elev", k = 2,
                 fit_args = list(chains = 2, iter = 1000))
  print(cv$overall)
  cv$predictive_coverage # coverage at 50/80/95% plus mean CRPS
}

## End(Not run)
```

cv_block_size_sweep *Cross-validate at a ladder of block sizes*

Description

A blocked cross-validation with blocks smaller than the autocorrelation range leaks: every held-out point has a near-identical neighbour in the training set, and the score is optimistic in proportion. The single number a `cv_*`() call returns cannot show this. This runs the same cross-validation at a ladder of block sizes, and by default once with random folds as the fully leaky reference. It returns the metric at each, with the estimated autocorrelation range alongside so that the curve can be read against it: it rises as the blocks pass the range and then plateaus, and the height of the rise is how much the random-fold number overstated the model.

Usage

```
cv_block_size_sweep(
  data_sf,
  response_var,
  predictor_vars,
  fit_fn,
  block_sizes = NULL,
  n_sizes = 6L,
  k = 5L,
  metric = "RMSE",
  include_random = TRUE,
  max_fits = 60L,
  sac = NULL,
  seed = 123L,
  quiet = FALSE,
  ...
)
```

Arguments

<code>data_sf</code>	An sf object with the response and predictors.
<code>response_var, predictor_vars</code>	Column names.
<code>fit_fn</code>	A function(<code>train_sf</code>) returning a <code>spatial_fit</code> , as for <code>cv_spatial()</code> ; see the example for wrapping a built-in backend.
<code>block_sizes</code>	Optional numeric vector of block edge lengths to sweep, in the CRS units the folds are built in (plain numbers; a units object is refused). Default NULL: the ladder described above. A size at which the grid would hold fewer than k blocks is not run, with a warning naming it and the largest size that still gives k blocks; if no size is left, the call is an error.
<code>n_sizes</code>	Number of sizes in the default ladder. Default 6.
<code>k</code>	Folds per cross-validation. Default 5.

metric	Which column of overall to read. Default "RMSE".
include_random	Also cross-validate with random folds, as the leaky reference. Default TRUE.
max_fits	The fit budget; see above. Default 60.
sac	Optional sac_range from <code>estimate_sac_range()</code> to mark on the curve, or a single number in the units of the CRS the folds are built in. An <code>estimate_sac_range()</code> result records the CRS it was fitted in; when that is not the sweep's, the range is converted to the sweep's units, with a warning. Default NULL: estimated here from the response, detrended on predictor_vars, when gstat is installed.
seed	Seed for the fold construction at every size.
quiet	Suppress the progress messages. Default FALSE.
...	Passed to <code>cv_spatial()</code> at every size (predict_args, p, parallel, metrics, ...). block_size, folds, k, seed and auto_range are set here and cannot be passed.

Value

A data.frame of class "block_size_sweep" with one row per cross-validation: block_size (NA for the random reference), method, blocks_used, k (the folds actually built), n_folds_succeeded, value (the pooled metric), fold_min, fold_max and fold_sd (its spread across folds). Attributes: metric, sac_range (the effective range, or NA), crs, k (the folds requested, which `print()` and `plot()` report), n_fits, response_var, and results, the full `cv_spatial()` result at every size. `plot()` draws it.

The fit budget

Each block size is a full cross-validation, so the cost is `length(block_sizes) * k` fits, plus `k` for the random reference. `max_fits` caps that (default 60: room for up to eleven sizes at `k = 5` plus the reference; the default six-size ladder at `k = 5` needs at most 35). A sweep that would run past the cap refuses to start, naming the number of fits it would have needed. Raise `max_fits` deliberately; the RF example below takes seconds, a Bayesian `fit_fn` takes minutes per fit.

The ladder

When `block_sizes` is NULL, `n_sizes` values are log-spaced from a twenty-fifth of the shorter side of the data's extent (of the longer side when the points lie on one line parallel to an axis) to the largest size, at most half that side, at which the grid still holds `k` blocks. Half the side gives a grid two blocks across, enough for `k` up to 4 and, for larger `k`, on an extent long enough in the other direction; on a roughly square extent at the default `k = 5` the top is about a third of the side. Any size at which the grid would hold more than the 1,000,000 `make_folds()` will build is dropped. The count is of grid cells: on clustered data a grid of `k` or more cells can have fewer than `k` that hold points, and `make_folds()` then lowers `k` at that size, which the `k` column shows. On an extent much longer than it is wide every rung can fall below the autocorrelation range while longer blocks would still fit `k` times along the longer side; the sweep warns when that happens, and `block_sizes` is then the way to reach past the range. Sizes are in the units of the CRS the folds are built in (`make_folds()`'s `params$crs`, metres for geographic input), and the returned table records that CRS. Each is the *minimum* block edge handed to `make_folds()`, which fits a whole number of cells across the extent, so the cells are somewhat longer than the size on the axis (up to twice as long).

See Also

[plot.block_size_sweep\(\)](#).

Other cross-validation: [area_of_applicability\(\)](#), [cv_bayes\(\)](#), [cv_gwr\(\)](#), [cv_rf\(\)](#), [cv_spatial\(\)](#), [estimate_sac_range\(\)](#), [fold_separation\(\)](#), [gwr_model_selection\(\)](#), [make_folds\(\)](#), [sac_nugget\(\)](#), [select_features_forward\(\)](#)

Examples

```
if (requireNamespace("ranger", quietly = TRUE) &&
    requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 200
  x <- 5e5 + runif(n, 0, 1000); y <- 5e6 + runif(n, 0, 1000)
  d <- as.matrix(dist(cbind(x, y)))
  field <- as.numeric(t(chol(exp(-d / 100) + diag(1e-6, n))) %*% rnorm(n))
  dat <- st_as_sf(data.frame(x = x, y = y, a = rnorm(n)), coords = c("x", "y"),
                  crs = 32632)
  dat$z <- field + 0.5 * dat$a + rnorm(n, 0, 0.2)
  rf_fn <- function(train_sf)
    fit_rf_model(train_sf, "z", "a", include_coords = TRUE, num_trees = 100, seed = 1)
  sw <- cv_block_size_sweep(dat, "z", "a", fit_fn = rf_fn, k = 4, n_sizes = 4,
                           quiet = TRUE)
  print(sw) # the curve as a table, with the random-fold reference
  if (requireNamespace("ggplot2", quietly = TRUE)) plot(sw)
}
```

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, with no 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,
pointsize = "auto",
block_size = NULL,
auto_range = FALSE,
parallel = FALSE,
metrics = NULL
)

```

Arguments

<code>data_sf</code>	An sf object.
<code>response_var</code>	Response column name.
<code>predictor_vars</code>	Predictor column names.
<code>folds</code>	Optional fold definitions, in any of three shapes: a make_folds() return value; a bare list of <code>list(train =, test =)</code> pairs of <code>..row_id</code> 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 (<code>blockCV::cv_spatial()</code> returns one as <code>\$folds_ids</code>), since its <code>\$folds_list</code> holds two <i>unnamed</i> 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. 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).
<code>k</code>	Number of folds. Default 5.
<code>seed</code>	RNG seed. Default 123.
<code>adaptive</code>	Logical; use adaptive bandwidth. Default TRUE.
<code>bandwidth</code>	Optional bandwidth, applied to every fold. For <code>adaptive = TRUE</code> an integer number of neighbours; for <code>adaptive = FALSE</code> 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 <code>GWmodel::bw.gwr()</code> , which is reported in <code>fold_metrics\$bandwidth</code> . See fit_gwr_model .
<code>kernel</code>	Kernel function type.
<code>boundary</code>	Optional polygonal sf/sfc for CRS alignment.
<code>pointsize</code>	Geometry coercion strategy.
<code>block_size</code>	Optional minimum block edge length for spatial CV blocks (projected CRS units). Blocks are then at least as large as the spatial autocorrelation range.
<code>auto_range</code>	Logical. If TRUE and <code>folds</code> is NULL, estimate the autocorrelation range and use it as the minimum block size. Default FALSE.
<code>parallel</code>	Accepted so that every <code>cv_*</code> () function takes the same arguments, but the GWR folds always run one after another in this R process. <code>GWmodel</code> is built with OpenMP, and OpenMP (GNU libgomp) deadlocks <code>parallel::mclapply()</code> 's

	forked workers once a GWR has been fitted in the session, for example by <code>fit_gwr_model()</code> , so forking would hang the call. A value asking for more than one core raises a warning saying so. Default FALSE.
metrics	Optional scoring function of your own, a function(<i>y</i> , <i>yhat</i>) returning a named numeric vector; its names become columns of <code>fold_metrics</code> (per fold) and overall (pooled) beside the built-in ones. See Your own metrics on <code>cv_spatial()</code> for the contract.

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`, `fold_status`, `orphan_rows`, `n_unknown_ids`, `n_dropped`, `formula` and `adaptive`. The two fold counts make a run where every fold failed visible in the return value itself, beyond the warning, since overall is a well-formed all-NA row either way, and `fold_status` (one row per fold: `fold`, `status`, `message`) says why each missing fold is missing. See `cv_spatial` for the five statuses and for `orphan_rows`. `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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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_block_size_sweep()`, `cv_rf()`, `cv_spatial()`, `estimate_sac_range()`, `fold_separation()`, `gwr_model_selection()`, `make_folds()`,

```
sac_nugget(), 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 = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * (st_coordinates(dat)[, 1] - 5e5) +
    2 * dat$elev + rnorm(n)
  cv <- cv_gwr(dat, "price", "elev", k = 3, bandwidth = 30)
  print(cv$overall)      # pooled over the held-out rows of every fold
  cv$fold_metrics        # and fold by fold
}
```

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,
  pointsize = "auto",
  metrics = NULL,
  ...
)
```

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 <code>list(train =, test =)</code> pairs of <code>.row_id</code> 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, since <code>blockCV::cv_spatial()</code> 's <code>\$folds_list</code> holds two <i>unnamed</i> vectors per fold and is refused by name. That function returns a label vector as <code>\$folds_ids</code> . 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. 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 <code>fit_fn</code> wrapping <code>fit_rf_model(seed =)</code> .
parallel	Passed to cv_spatial . Default FALSE. Under forked workers each fold's forest runs single-threaded unless <code>num_threads</code> is passed explicitly, so <code>parallel = 4</code> means four threads in total, where it would otherwise mean four times the session's <code>mc.cores</code> .
block_size, auto_range, boundary	Passed to cv_spatial .
pointsize	How non-POINT geometry is reduced to a point before fitting; passed to cv_spatial . Default "auto".
metrics	Optional scoring function of your own, passed to cv_spatial ; see Your own metrics there.
...	Passed to fit_rf_model on every fold: <code>num_trees</code> , <code>mtry</code> , <code>importance</code> , <code>include_coords</code> and so on. <code>data_sf</code> , <code>response_var</code> , <code>predictor_vars</code> and <code>.already_prepped</code> are set by this function and must not be passed here (every fold would fail with "matched by multiple actual arguments"). <code>seed</code> is this function's own argument and never reaches <code>fit_rf_model()</code> through here; see <code>seed</code> above for growing every fold's forest from one fixed seed.

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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows

that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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_block_size_sweep()`, `cv_gwr()`, `cv_spatial()`, `estimate_sac_range()`, `fold_separation()`, `gwr_model_selection()`, `make_folds()`, `sac_nugget()`, `select_features_forward()`

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 200
  dat <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + 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,
    metrics = NULL,
    .caller = "cv_spatial"
  )

```

Arguments

<code>data_sf</code>	An sf object.
<code>response_var</code>	Response column name.
<code>predictor_vars</code>	Predictor column names.
<code>fit_fn</code>	A function of one argument, the training slice of <code>data_sf</code> , returning a <code>spatial_fit</code> built with <code>new_spatial_fit()</code> . 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 <code>predict.<subclass>()</code> method registered, because that is how each fold is scored.
<code>folds</code>	Optional fold definitions, in any of three shapes: a <code>make_folds()</code> return value; a bare list of <code>list(train =, test =)</code> pairs of <code>..row_id</code> 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 (<code>blockCV::cv_spatial()</code> returns one as <code>\$folds_ids</code>), since its <code>\$folds_list</code> holds two <i>unnamed</i> 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. IDs naming no row in the prepared data are dropped with a logged count. Built via <code>block_kfold</code> when <code>NULL</code> .
<code>k</code>	Number of folds.
<code>seed</code>	RNG seed.
<code>boundary</code>	Optional boundary for fold construction.
<code>pointize</code>	Geometry coercion strategy.
<code>predict_args</code>	Extra arguments for <code>predict()</code> .
<code>fold_info_fn</code>	Optional function(<code>fit</code> , <code>test_sf</code> , <code>y</code> , <code>yhat</code>) returning a named list (or a named vector) of per-fold extras (a bandwidth, a tuning value, anything read off the fitted object), added as columns of <code>fold_metrics</code> . It sees the fit and the held-out layer, which <code>metrics</code> does not; it is applied per fold only, and its values are not pooled. An element <code>..per_row</code> that is a data frame with one row per held-out observation is spliced into predictions instead (NA in the rows of a fold that returned none; one of the wrong length is dropped and logged); its columns must

	be named, once, and not reuse a column predictions already has (<code>.row_id</code> , <code>fold</code> , <code>y</code> , <code>yhat</code> , <code>y_train_mean</code>). Every other element must be named, hold one value, and not reuse a column <code>fold_metrics</code> already has; anything else is an error. A <code>fold_info_fn</code> that throws on a fold is logged, its columns are NA for that fold, and <code>fold_status\$message</code> says so; the fold is kept.
<code>p</code>	Number of predictors for Adj R^2 (NULL to skip). Only meaningful for models with a fixed global parameter count; pass NULL for models with spatially varying coefficients (e.g. GWR).
<code>block_size</code>	Optional minimum block edge length for spatial CV blocks (projected CRS units).
<code>auto_range</code>	Logical. If TRUE and <code>folds</code> is NULL, estimate the autocorrelation range and use it as the minimum block size. Default FALSE.
<code>parallel</code>	Logical or positive integer. If TRUE, auto-detect the number of cores and fit folds in parallel via <code>parallel::mclapply()</code> (macOS / Linux; falls back to sequential on Windows). If an integer > 1, use that many cores. Default FALSE (sequential). A learner that runs OpenMP code (GWmodel, or an xgboost built with GNU libgomp) can hang the forked workers once it has run in the session; keep such a <code>fit_fn</code> sequential, as <code>cv_gwr()</code> does.
<code>metrics</code>	Optional scoring function of your own; see Your own metrics below. Default NULL: the built-in metrics only.
<code>.caller</code>	Internal. The name the messages carry, so a wrapper such as <code>cv_rf</code> reports itself in place of <code>cv_spatial()</code> .

Value

A list with overall, `fold_metrics`, `predictions`, `folds`, the two fold counts `n_folds_attempted` and `n_folds_succeeded`, and four elements that say what happened to the difference between them and to the rows: `fold_status`, `orphan_rows`, `n_unknown_ids` and `n_dropped`. 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. `fold_status` is a data.frame with one row per fold supplied (fold, status and message), where status is "ok" (message is empty unless `fold_info_fn` failed on the fold); "error" (the fit or its `predict()` threw; message is the error text); "skipped" (nothing scorable: too few matched rows, a prediction of the wrong length, or no finite observed/predicted pair); "dropped" (an empty test set, or fewer than two training rows, once incomplete rows were removed, so the fold never reached the fitter); or "worker_error" (a parallel worker died, for example killed for lack of memory). Each fold runs in its own worker, so a failure costs that fold only, and an error that stops a sequential run (a `metrics` or `fold_info_fn` return value of the wrong shape) stops a parallel one too, naming the fold. Every fold missing from `fold_metrics` has its reason there, which matters most when the console output of a long run is gone. When some folds, but not all, end as "error", "skipped" or "worker_error", the function warns, naming them and how many rows overall covers: it is pooled over the folds that produced predictions, and the fold that fails is often the hardest to predict, so it may flatter the model. (A "dropped" fold has its own warning.) `orphan_rows` holds the `.row_ids` of rows in the data that no fold names (they enter no training set and are never scored; non-empty only when the folds were built on a different or subsetted layer), and `n_unknown_ids` counts the distinct row IDs the folds name that the data does not have. Each such row is named by every fold, once as a test row and once in each other fold's training set, and this counts the row, not the mentions (expected when rows

were removed for missing values). `n_dropped` is how many rows `prep_model_data()` removed for missing or non-finite values or a bad geometry before any fold was fitted. The `fold` column of `fold_metrics`, `predictions` and `fold_status` 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. Splits that already carry a `fold_id`, as the folds of a `cv_*`() result do, keep it: handing one run's folds to another labels each fold as the first run and `fold_separation()` do, a dropped fold's gap included. R^2 is out-of-sample R^2 : the total sum of squares is taken about the mean of the *training* rows (in overall, each held-out row about its own fold's training mean), the null prediction available when the fold is predicted, and not about the held-out rows' own mean, a null model that would know the test data. On spatial blocks the two can differ widely; R^2 is below 0 when the model predicts worse than the training mean. `model_metrics(newdata = )` uses the same baseline. `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.

Name collision with blockCV

The **blockCV** package exports a function of the same name that does the opposite job: `blockCV::cv_spatial()` *builds* spatial folds, where this function *runs* a cross-validation over folds it is given. With both packages attached, whichever was attached last masks the other; `spatialkit::cv_spatial()` always resolves to this one. The two cooperate: `blockCV::cv_spatial()` returns its fold assignment as `$folds_ids`, a vector of fold labels, and that vector is accepted directly as the `folds` argument here and in every other `cv_*`() function. Fold construction is **blockCV**'s home ground and this package does not try to match its breadth there; what this package adds is the path from irregular points through data-drawn regions to a cross-validated, compared model.

Your own metrics

The built-in columns (RMSE, MAE, MAPE, SMAPE, R^2 , Adj_R^2) are the Gaussian regression set, and the section below says which of them survive a count or a bounded response. `metrics` is the way to score what they cannot: a function(y , $\hat{y}$) that returns a named numeric vector (a named list of scalars, or a one-row data frame, also serve), for example a Poisson deviance, a log score on a probability, or a loss with your own weights. It is applied twice, in the same way the built-in metrics are: once per fold, to that fold's held-out rows, so each name becomes a column of `fold_metrics`; and once to the pooled out-of-sample predictions of every fold, so each name becomes a column of `overall`. Only the pairs the built-in metrics use reach the function (both y and $\hat{y}$ finite), so its columns describe the same rows as RMSE, and `n_pred` counts them.

The contract: every element named, names unique and not one of the built-in column names, one number per name. The built-in names include the per-fold extras of the backend or of your `fold_info_fn` (bandwidth for `cv_gwr()`; CRPS, coverage_*, `gp_k`, `gp_n_basis` and `n_draws` for `cv_bayes()`), and `mean_CRPS`, which `compare_models_cv()` writes. Anything else is an error, because a scoring function that returns the wrong shape is a mistake to surface instead of a fold to skip. A function that *throws* on a fold is logged and its columns are NA for that fold (and for overall, if it throws on the pooled predictions); a fold is never dropped for it. When no fold produced a prediction the empty `fold_metrics` frame still carries the function's columns, typed, provided the function can be called on zero-length input.

`fold_info_fn` is the per-fold half of the same mechanism, with access to the fitted object and the held-out layer; `metrics` sees only the two vectors but is also pooled. `compare_models_cv()` hands

one metrics to every backend, so the columns are comparable across the rows of its overall.

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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The n_MAPE and n_SMAPE columns record how many rows that was; the n column counts finite observation/prediction pairs. Read a percentage error next to its count: when n_MAPE < n, MAPE is an average over a subset of the data, whatever its value.

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, which n_MAPE = 58 now says. 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; n_SMAPE shows how many rows it kept, and the count is only a label, not a repair.

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_block_size_sweep()`, `cv_gwr()`, `cv_rf()`, `estimate_sac_range()`, `fold_separation()`, `gwr_model_selection()`, `make_folds()`, `sac_nugget()`, `select_features_forward()`

Examples

```
library(sf)
set.seed(1)
n <- 80
site <- st_as_sf(
  data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
             elev = rnorm(n)),
  coords = c("x", "y"), crs = 32632
)
site$price <- 10 + 0.01 * (st_coordinates(site)[, 1] - 5e5) +
  2 * site$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$overall
# Compare these before trusting the metrics above; fold_status says why
# any fold is missing from fold_metrics.
c(attempted = cv$n_folds_attempted, succeeded = cv$n_folds_succeeded)
cv$fold_status

# 3. A metric of your own beside the built-in ones: per fold and pooled.
med_ae <- function(y, yhat) c(MedAE = stats::median(abs(y - yhat)))
cv2 <- cv_spatial(site, "price", "elev", fit_fn = lm_fit, k = 3, seed = 1,
  metrics = med_ae)
cv2$overall$MedAE
cv2$fold_metrics[, c("fold", "RMSE", "MedAE")]

```

determine_optimal_levels

Determine an optimal number of spatial levels via an elbow heuristic

Description

Computes a WSS curve over $k=1..K_{\text{max}}$ 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"),
  select_on = c("all", "split")
)

```

Arguments

data_sf	An sf object. Features with empty or non-finite coordinates are dropped with a warning.
max_levels	Integer upper bound on levels. Default 12. The sweep also stops at the number of distinct locations (k-means cannot place more centres) and one short of the number of points.
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 for the subsample and the k-means++ restarts; restored afterwards. Default 123. The rows are put in coordinate order before either, so the answer does not depend on the order they come in.
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 and is never coerced, because the residuals of an OLS fit to arbitrary level codes carry no meaning to test for autocorrelation. Rows where it, or a predictor, is missing or non-finite stay in the WSS sweep and the cells and are left out of Moran's I; a logged warning gives their number.
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 <i>significant</i>), or "combined" (rank-average of the WSS curve's log-log sag, the quantity the elbow is read from, and that same significance). Falls back to "geometric", with a warning, when response_var or predictor_vars is not given, and with a logged warning when no candidate clears the nine-cell resolution floor described in Details . A response_var or predictor_vars naming a column that is not in data_sf is an error. Supplying both response_var and predictor_vars upgrades "geometric" to "combined": the selection then depends on the response (see "Post-selection inference").
select_on	"all" (default) selects on every point; "split" reads the response on one spatially blocked half of the points only and returns the other half as the set to estimate on, so that the standard errors computed downstream on the chosen cells are not post-selection. The count is still chosen for the whole layer. See "Post-selection inference".

Details

The elbow is read on log-log axes, and there may be none. Points with no cluster structure have a WSS curve close to c/k , and the classical rule, the point furthest below the chord from the first to the last k, still finds a "knee" on it on linear axes, at about $\sqrt{K_{max}}$: 4 at the default max_levels = 12 and 13 at 160, whatever the data. On $\log k$ against $\log$ WSS that curve is a straight line, while separated clusters fall faster than it until there is one cell per cluster and like it after, a bend at the cluster count. The elbow is therefore the k whose $\log$ WSS sags furthest below the straight line

joining $k = 1$ and K_{max} on those axes, and it counts as one only when the sag is at least $\log 1.25$ (the WSS a fifth below the power law through the ends). Measured on 60 to 1500 points with ladders to 3–40, uniform layouts over squares, discs, triangles, an L-shape, density gradients and jittered lattices sagged at most 0.16, and two to ten separated clusters at least 0.6 once `max_levels` passed the cluster count. With no elbow the function warns and returns the linear-axis answer, which the ladder chose, not the data. An elongated extent also bends, at about its aspect ratio, because the first cuts go across its long axis (0.2–0.45 for a 4:1 rectangle); past the threshold that bend is reported as an elbow, and it describes the extent’s shape rather than clusters in it. When locations repeat (stations visited many times), the sweep can reach one cell per distinct location, where the WSS is zero (to within 10^{-12} of the total). That k has no place on log axes and is left out of the line, but the fall to zero is the sharpest bend there is: when the rest of the curve has no elbow, the number of distinct locations is the elbow.

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. It is a direct reflection of the spatial process being modeled instead of the 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 $\pm \text{top}_n$ if larger), so that only the most promising candidate k values incur the cost of the full Moran’s I computation.

The WSS curve is read for its shape, so it is fitted to be smooth. Every point on it is a k -means local optimum, and with a few random restarts the curve mixes level effects with optimisation noise: measured on clustered layouts, a sweep over $k = 1 \dots 30$ at `stats::kmeans(nstart = 5)` rose at one or two steps in three of five draws, and an earlier form of the elbow rule once selected such a bump. Each k is therefore fitted as the best of 25 restarts seeded by `k-means++` (Arthur and Vassilvitskii 2007), the budget at which the gain from further restarts saturates (Fränti and Sieranoja 2019; Steinley 2003 on why the usual handful is not enough). The same sweeps then had no increase at all. A curve that still rises somewhere is reported but not refused: a bumpy curve is uncertain, not unidentified. A logged warning names the number of rising steps, and the model-aware paths return it as `wss_bumps` in the “diagnostics” attribute beside `wss_spread`, the relative spread of WSS across the restarts at each k . Because the optimiser changed, a selection made by an earlier version on a curve that had such a bump can differ from the one made now; where the earlier curve was clean, the selection is the same.

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 $|z| = |I - E[I]|/sd(I)$ using the Cliff & Ord regression residual moments. The cell-level residuals are OLS residuals by construction, which is the case those moments are derived for, but the derivation also assumes errors of equal variance, and a cell mean over n_j points has a variance proportional to $1/n_j$. Over the same runs z had mean ≈ 0 , $sd \approx 1$ and a two-sided 5% rejection rate of 0.040–0.057 at every k , and it stayed calibrated on gradient and moderately clustered layouts; with single-point cells next to cells of 70 or more points its mean rose to 0.2–0.34 and its rejection rate to 7–8% at 20

cells. Where structure remains, $|z|$ mixes its size with the number of cells it is measured on, since $\text{sd}(I)$ shrinks as cells are added. 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 $\text{Var}[I] = 0$ 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, the whole call falls back to the geometric ranking and logs a warning that says so. That is the usual outcome well past `max_levels = 10`: the neighbourhood is the elbow plus or minus `max(4, top_n)`, and on points with no cluster structure the elbow sits near $\sqrt{K_{max}}$, so the neighbourhood reaches ten cells only from about `max_levels = 40`. Measured on 1000 uniform points, `max_levels` of 12, 20 and 30 all fell back, and 40 scored $k = 10$ and 11 alone. On such a layer `resolution_profile()`, which scores Moran's z at every level of its ladder, is the model-aware view. Under `criterion = "combined"`, an elbow below ten cells is a count Moran's I cannot score, so it cannot weigh the response there: the geometric ranking is returned, with a logged warning, and the diagnostics record it (see `Value`). (Ranking the window anyway put the smallest count Moran's I scores, ten, first whatever the response did.) Otherwise the candidates below the floor that sit alongside candidates above it all take the last place on the Moran's I axis, after every candidate it scored, while still competing on the geometric axis. That axis is the elbow's own log-log sag at each candidate; when the WSS curve has no elbow it is flat, every candidate tied, and Moran's I alone orders them.

"combined" is not an estimate of the number of clusters. Where the elbow is at ten cells or more, Moran's I can move the pick away from it when the response is still spatially structured at the elbow's resolution. Use "geometric" when the cell count should follow the clustering of the points, and `resolution_profile()` when the response should drive it.

Value

An integer vector of candidate level counts, **best first**: under the geometric criterion the elbow, then its lower and upper neighbours (with a warning when the WSS curve has no elbow and the first is the ladder's choice; see `Details`); 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`), the WSS curve (`wss`) with the relative between-restart spread at each k (`wss_spread`), the number of rising steps on it (`wss_bumps`), the restart budget (`nstart`), the geometric elbow the evaluated neighbourhood was drawn around (`knee_k`), the k at which k -means failed (`failed_k`; their `wss` entries are interpolated from the neighbours, not measured) and the k the model-aware pass actually scored (`eval_ks`, the elbow's neighbourhood). Under "combined" it also carries the WSS of the re-run clustering at those k (`wss_eval`), the rank average that ordered them (`combined_rank`, named by k) and `criterion = "combined"`; when "combined" returned the geometric ranking because the elbow is below ten cells, it carries `criterion = "geometric"` and `fallback` (the reason) in place of `combined_rank`. 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), no diagnostics are available

and the attribute is absent. Both fallbacks are logged as warnings. The geometric path returns a plain integer vector; a rising WSS curve is still logged there. With `select_on = "split"` every path adds a "split" attribute: a list with selection and estimation (integer row positions in `data_sf`), method and seed. For a full per-level table of criteria, cell support, restart spread and the flat region, see `resolution_profile()`.

Post-selection inference

When the selection reads the response (here, whenever both `response_var` and `predictor_vars` are supplied), everything estimated afterwards on the chosen cells is estimated on data that already influenced the choice, and its standard errors are post-selection ones: descriptive, not at nominal coverage (Gao, Bien and Witten 2022; Chen and Witten 2023 give the exact selective test for two k-means clusters, the first link of this chain). The exposure is narrower than "the partition was chosen on the response": every partition here is k-means on the coordinates alone, and the response only decides which *count* is ranked first. It is not zero, because the count determines every cell the downstream standard errors are computed over.

`select_on = "split"` is sample splitting: the layer is cut into two spatially blocked halves (`make_folds(k = 2, method = "block_kfold")`), the criteria that read the response (Moran's I on the cell means) read the first half only, and the row positions of both halves come back in the "split" attribute (selection and estimation). The WSS curve and the k-means cells still use every point: they read coordinates alone, and the count is for a tessellation of every point, so it is chosen on that layer's extent and clusters rather than on half of them. Build the tessellation on every point (cells are geometry), but aggregate and fit on `data_sf[attr(x, "split")$estimation, ]`, whose response the selection never saw. That keeps the selection's use of the response out of the estimates, but only as far as the two halves are independent: the split has no buffer, so points near the border between them are correlated with the selection half over the autocorrelation range, and coverage is nominal only when that range is short against the blocks. The price is precision: half the points estimate, and García Rasines and Young (2023) show a *contiguous* spatial half is less efficient than the exchangeable split the i.i.d. theory assumes, because the two halves are not interchangeable. Two alternatives keep the whole sample: data thinning for count responses (Neufeld et al. 2024) and data fission for Gaussian-like ones (Leiner et al. 2023). Neither is implemented here; the split needs no distributional assumption, which is why it comes first. Selection on coordinates alone ("geometric" with no response) is not exposed in this way, and "split" then changes nothing but the attribute.

The estimation rows cover only the estimation half's blocks, while the cells cover the whole layer, so the estimation rows do not fill every cell. A cell inside the selection half gets none and comes back NA; a cell across the border between the halves is estimated from its estimation-half points alone, and its standard error describes that part, not the cell (on 400 simulated fields a nominal 95% interval covered 0.97 in cells inside the estimation half and 0.88 in cells across the border). Count, per cell, how many of its points are estimation rows, and read inferential results only off cells whose points all are.

References

- Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of careful seeding. *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1027–1035.
- Fränti, P. and Sieranoja, S. (2019). How much can k-means be improved by using better initialization and repeats? *Pattern Recognition*, 93, 95–112. doi:10.1016/j.patcog.2019.04.014

- Steinley, D. (2003). Local optima in K-means clustering: what you don't know may hurt you. *Psychological Methods*, 8(3), 294–304. doi:10.1037/1082989X.8.3.294
- Gao, L. L., Bien, J. and Witten, D. (2022). Selective inference for hierarchical clustering. *Journal of the American Statistical Association*, 119, 332–342. doi:10.1080/01621459.2022.2116331
- Chen, Y. T. and Witten, D. M. (2023). Selective inference for k-means clustering. *Journal of Machine Learning Research*, 24(152), 1–41. <https://jmlr.org/papers/v24/22-0371.html>
- García Rasines, D. and Young, G. A. (2023). Splitting strategies for post-selection inference. *Biometrika*, 110(3), 597–614. doi:10.1093/biomet/asac070
- Leiner, J., Duan, B., Wasserman, L. and Ramdas, A. (2023). Data fission: splitting a single data point. *Journal of the American Statistical Association*, 120(549), 135–146. doi:10.1080/01621459.2023.2270748
- Neufeld, A., Dharamshi, A., Gao, L. L. and Witten, D. (2024). Data thinning for convolution-closed distributions. *Journal of Machine Learning Research*, 25(57), 1–35. <https://jmlr.org/papers/v25/23-0446.html>

See Also

`build_tessellation()` and `get_voronoi_seeds()`, which accept this function's result directly as `approx_n_cells` and `n` (the first candidate is used, and the output records that it came from here); `assign_features_to_polygons()` and `summarize_by_cell()` for the steps that follow.

Other aggregation: `assign_features_to_polygons()`, `kriging_adequacy()`, `resolution_profile()`, `select_resolution()`, `summarize_by_cell()`, `summary.resolution_profile()`

Examples

```
library(sf)
set.seed(1)
# Two clearly separated clusters: the elbow should sit near k = 2
pts <- st_as_sf(
  data.frame(x = 5e5 + c(runif(25, 0, 10), runif(25, 90, 100)),
             y = 5e6 + 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, purpose = c("distance", "area"))
```

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 <code>sf::st_crs()</code> ; an unusable value (one that resolves to NA_crs_) raises an error, so x is never left silently unprojected.
purpose	Which property the projection is for. "distance" (the default, and everything above): the candidate that distorts pairwise distances least, which is what ranges, block sizes, bandwidths and length-scales read off the coordinates. "area": densities or rates per cell are going to be computed, so the CRS must be equal-area. For lon/lat input the choice is then made among equal-area projections only: a Lambert azimuthal centred on the data, or an Albers conic where its parallels do not degenerate, whichever distorts distances less. A UTM zone (conformal, not equal-area) never enters that comparison, and global coverage gets Equal Earth in place of Web Mercator. Already-projected input is still returned untouched, but its area distortion over the extent is measured (the spread of planar-to-geodesic area ratios over probe polygons) and logged as a warning when it exceeds 1 percent. Measured: a zone's own width edge to edge, 0.25 percent; the conterminous United States forced into one zone, 14 percent; Web Mercator over 2.5 degrees of latitude at 48N, 4 percent; an equal-area projection, a few tenths of a percent, which is the sphere the geodesic areas are computed on against the ellipsoid the projection uses. <code>summarize_by_cell()</code> applies the same measurement before it computes a density. Ignored when target_crs is given.

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. The centre is the centroid on the sphere, computed with s2 whatever `sf::sf_use_s2()` is set to, so data near a zone edge get the same zone in every session.

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 one point per feature, plus the vertices of its outline when it has fewer than 40 features, so a single study-area polygon is scored too) 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 $\pm 180^\circ$ 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.

Around a pole Data spanning more than 180 degrees of longitude with no such gap surround a pole. When every point also lies on one side of the equator (Antarctic stations, a pan-Arctic network), a Lambert azimuthal equal-area centred on that pole is used, provided it measures a smaller distance error than the global fallback. Web Mercator splits such a layer at +/-180 degrees and stretches it towards the pole: a ring of Antarctic stations measured a worst-case distance error near 20,000 percent in it, against about 2 percent in the polar projection. Only the coverage left over, spanning both hemispheres or a low-latitude belt the polar projection fits worse, falls back to EPSG:3857 (Equal Earth for purpose = "area").

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**. That is 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. When a projection was chosen here (lon/lat input, no `target_crs`) the result carries `attr(x, "crs_choice")`: a data.frame with one row per projection considered, holding name, crs (its definition), the measured worst-case distance_error (relative, over sampled pairs; NA where it could not be measured) and chosen. The figure the log line quotes for the winner is therefore recoverable for every candidate, and is measured for the single candidate on the paths where no comparison runs (a UTM zone on a local extent, the equal-area projection chosen for a layer straddling the antimeridian). It is NULL exactly when no local projection was chosen here: input that already carried a projected CRS, a `target_crs` you supplied, or an extent no local projection fits, which falls back to Web Mercator or Equal Earth. 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.

See Also

Other spatial data preparation: `clip_target_for()`, `coerce_to_points()`, `harmonize_crs()`, `prep_model_data()`

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; the zone's measured
# distance error over the extent travels with the result.
```

```

st_crs(ensure_projected(pts_ll))$epsg # 32632
attr(ensure_projected(pts_ll), "crs_choice")

# 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

# For densities per cell the CRS has to be equal-area. Two candidates are
# scored, an Albers conic and a Lambert azimuthal, both centred on the data,
# and whichever distorts distance less over the extent is used: here Albers.
st_crs(ensure_projected(pts_ll, purpose = "area"))$proj4string

```

`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

<code>polygons_sf</code>	An sf or sfc object containing polygonal features.
<code>id_col</code>	Character scalar; name of the identifier column.
<code>method</code>	One of "centroid", "surface_point", "bbox_center".
<code>make_valid</code>	Logical; apply <code>st_make_valid()</code> first. Default TRUE. The copy the sort key is measured on is repaired as well, after it has been transformed (see Details); with FALSE neither is.

`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 does not usually reverse two neighbouring cells. It can where two cells' centres lie within about that step of the same longitude, as fine cells stacked north-south near a projection's central meridian do: 36 of 2,500 100 m cells straddling a UTM central meridian changed ID after a transform to EPSG:3035. No rounding step removes that, so to match cells computed in different projections, join them on geometry rather than on the ID. The key is computed on the sphere (s2) whether or not `sf::sf_use_s2()` is on, so the session setting does not change the IDs. Set to NULL to sort in the input CRS, which gives IDs that are reproducible but not comparable across projections.

Details

A feature that is valid in its own CRS is not always one s2 can measure. Only the vertices are transformed to the sort CRS, and on the sphere they are joined by great-circle arcs, so a long edge that is straight in the layer's CRS and passes about a metre from another vertex of the same ring can end up on the other side of that vertex. The ring then crosses itself, and s2 refuses it (1 of 291 Voronoi cells of Texas clipped to the state outline in EPSG:5070; "Loop 1 is not valid: Edge 36 crosses edge 52"). The repair of the sort copy does not split crossing edges, so a feature s2 still refuses is handled on its own. Its sort copy is repaired a second time with the crossing edges split. If s2 refuses that as well, or with `make_valid = FALSE`, which asks for no repair, the feature's centroid and area are measured as plane geometry in the layer's own CRS and the centroid is transformed to the sort CRS. With the methods "surface_point" and "bbox_center" only the area needs this, because those points are not taken with s2.

The function warns once, saying how many features took each route. Their keys are close to the spherical ones and not equal to them. The second repair moved the Texas cell's area by about 40 square metres in 2,949 square kilometres. Its centroid taken in the plane lay 20 m from the one the second repair gave on the sphere, and the two kinds of centroid lay 5 m apart at the median and 61 m at most for the other 290 cells. A feature measured in the plane can therefore sort on the other side of a neighbour whose centre is that close to its own in longitude. Every other feature's key is unaffected, and the geometry returned is never the sort copy.

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.

See Also

Other tessellation: [build_tessellation\(\)](#), [create_grid_polygons\(\)](#), [create_grid_polygons_cached\(\)](#), [create_voronoi_polygons\(\)](#), [get_voronoi_seeds\(\)](#), [plot_tessellation_map\(\)](#), [voronoi_seeds_kmeans\(\)](#), [voronoi_seeds_random\(\)](#)

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 and re-derive the IDs: the SAME cell gets the same ID,
# which is the property a row-position ID does not have. The check joins
# the two layers on geometry, because comparing sorted ID vectors would
# pass for any two permutations of 1:9.
fwd <- ensure_stable_poly_id(g)
rev <- ensure_stable_poly_id(g[nrow(g):1, ])
same_cell <- match(st_as_text(st_geometry(rev)), st_as_text(st_geometry(fwd)))
all(rev$poly_id == fwd$poly_id[same_cell])
```

estimate_sac_range	<i>Estimate the spatial autocorrelation range from data</i>
--------------------	---

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 % of the sill; for the spherical model (fitted only when the exponential fit is singular or does not converge) 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,
  detrend = c("ols", "reml"),
  reml_max_n = 400L,
  keep_directional_fits = FALSE
)
```

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 <code>gstat::variogram()</code> 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. For a count or other response whose variance tracks its mean, see the section on non-Gaussian responses: the range is still estimated, but it is a less reliable number than for a Gaussian response.
predictor_vars	Optional character vector. When supplied, the trend on these predictors is removed first and the variogram describes the residual autocorrelation, the part a spatial model has to handle once the covariates have done their work. How the trend is removed is set by <code>detrend</code> , and it matters: see "Detrending and the residual-variogram bias". Rows whose response or predictor is missing or infinite are left out of that fit, and so of the variogram, with a logged count.
n_max	Maximum number of points to subsample before fitting. Variogram estimation is $O(n^2)$ so this keeps runtime bounded.
cutoff	Fraction of the maximum inter-point distance to use as the variogram lag cutoff. That distance is the farthest pair, found on the convex hull, and not the bounding-box diagonal, which depends on how the axes are oriented. Default 0.5.
range_frac	Positive numeric. A fitted range exceeding <code>range_frac * cutoff * max_dist</code> (that is, beyond the longest lag the empirical variogram was actually fitted over) is treated as unidentified and <code>NA_real_</code> is returned. <code>gstat::fit.variogram()</code> yields a finite number even when the variogram never reaches a sill, and such a value extrapolates past the observed lags instead of measuring a long autocorrelation range. Passing it to <code>make_folds(auto_range = TRUE)</code> would collapse the block grid to a single block. Default 1.0; raise it to accept ranges extrapolated beyond the fitted lags. The bound does not guarantee room for two blocks: at the defaults it is half the farthest-pair distance, about 0.71 of the side of a square layer, and a block grid needs a range below half the width of the bounding box in one direction or the other. An accepted range between the two leaves <code>make_folds(auto_range = TRUE)</code> room for a single block of that size (see its <code>auto_range</code> argument for what it does then). On a 1000 m square with an exponential field of effective range 570 ($n = 300$), 9 of 30 draws were accepted in that band. Lowering <code>range_frac</code> to fit the grid would turn those estimates into NA and the blocks into geometric ones smaller than the range.
seed	RNG seed for the <code>n_max</code> subsample, restored afterwards so the caller's random stream is untouched. Default 123L: the subsample is an internal approximation and no 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 <code>nrow(points_sf) <= n_max</code> , where nothing is sampled. The <code>reml_max_n</code> subsample uses the same seed.
detrend	How the trend on <code>predictor_vars</code> is removed; ignored when there are none. "ols" (default) fits it by ordinary least squares and fits the variogram to the residuals. This is the long-standing behaviour, which underestimates the range (see the section below). "reml" fits the trend and an exponential-plus-nugget covariance together by residual maximum likelihood with <code>nlme::gls()</code> , returns the REML range, and attaches the empirical variogram of the REML residuals for inspection. It needs nlme , costs $O(n^3)$ (about 2 s at 300 points, 12 s at

	500, 45 s at 800), and so runs on at most <code>reml_max_n</code> points; when it does not converge the "ols" path runs instead with an R warning saying so.
<code>reml_max_n</code>	Positive integer, at least 30. With <code>detrend = "reml"</code> , the trend and covariance are fitted on a seeded random subsample of this many points when the layer has more (exact duplicate locations are dropped first); the fitted trend is then removed from every point. Default 400. Raise it for a better-determined fit at the cost above.
<code>keep_directional_fits</code>	Logical. Attach each direction's empirical variogram and fitted model as <code>directional_fits</code> ? Defaults to FALSE: the four variograms are most of the object's size (42.1 KB of 59.3 KB at $n = 400$, and the difference between a 52.4 KB and a 94.6 KB <code>make_folds(auto_range = TRUE)</code> result), while the numbers read from them (<code>directional</code> , <code>directional_fitted</code> , <code>directional_status</code> , <code>anisotropy</code>) are attached either way, and <code>plot()</code> draws the effective variogram from its own attribute. Set TRUE to inspect the directional curves.

Details

The exponential model is kept whenever it converges, without comparing it with the spherical fit, and on fields smoother than exponential that makes the range long. Measured on simulated fields ($n = 300$ on a 1000 m square, 30 draws each): about 1.8–2.1 times the practical range of a Gaussian covariance, and 1.3–1.4 times the range of a spherical one, while an exponential field came back at 0.97 of its effective range. The error is on the safe side (blocks too large, cross-validation pessimistic), and it is kept on purpose: choosing the family by the smaller weighted sum of squares corrects the spherical case but sends exponential fields low, to about 0.82 of the truth, which is the direction that leaks.

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 $\pm 22.5^\circ$ tolerance. Those four windows 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% above the truth, and no hurdle placed in front of it (all four directions fitted, ratio above 1.5, maximum above $1.5 \times$ 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 auto-correlation range to avoid leakage, and the conservative choice is to size them from the longest directional range explicitly. Read it from `directional_fitted`, not `directional`: on a strongly anisotropic field the major axis is the direction most likely to run past the fitted lags, which leaves it NA in `directional`, so `max()` of that is NA, or with `na.rm = TRUE` the second-longest range. Check `directional_status` first: a major axis marked "over_cutoff" has no identified range at all, and a longer cutoff or `make_folds(method = "nndm")` is the way on. A ratio above 1.5 is written to the package log at INFO level with that advice, which reaches the session log file but not the console (the ratio passes 1.5 on most isotropic fields too); `print()` shows the directional ranges and the ratio, and `attr(range, "anisotropy")` holds it. Only when the omnidirectional fit is singular or did not converge is the directional maximum returned in its place, and `anisotropy_used` is TRUE in

that case alone. An omnidirectional fit that converged to a range past the fitted lags is refused (see the Value section) whatever the directions found: the directions that reached a sill are the shorter ones, so their maximum is a lower bound, not an estimate.

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/h^2 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.

The lags are binned the way **gstat** bins them by default, 15 bins out to the cutoff, each `cutoff * max_dist / 15` wide (about 47 m on a 1000 m square at the defaults). A range spanning only one or two bins is resolved coarsely and comes out long: exponential fields with an effective range of 60 m ($n = 300$ on a 1000 m square, 30 draws) returned a median of 89–102 m, where the same fields binned over a 200 m cutoff gave 65–68, and at a range of 300 m there was no bias. A range shorter than the first bin cannot be resolved at all and can come back several times too long: an effective range of 24 m ($n = 1500$ on a 1000 m square, 8 draws) returned 93–479 m, five of them as the directional maximum, against 19–32 m at `cutoff = 0.1`; the first bin's semivariance was 92–99 percent of the fitted sill in all eight. So whenever the empirical variogram is already at its sill in the first one or two bins (`plot()` the result), run it again with a smaller cutoff, whatever range was fitted.

Nothing tests whether the layer has spatial structure at all. On white noise ($n = 300$ on a 1000 m square, 30 draws) the estimate was a finite, spurious range (57–533 m) in 8 draws and a refusal in the rest, mostly as past the fitted lags or not converged, and only once as no model fitted; with `detrend = "rem1"` it was finite in 13 of 30 (21–453 m), and 16 of the refusals were ranges of 0.18–12.5 m, too short for 30 pairs of points to lie inside them. A spurious range errs towards larger blocks, so the harm is mostly lost training data, but a caller who needs to know whether there is any structure should look at the variogram (`plot()` on the result) rather than at whether the answer is NA.

When the all-pairs fit is singular or did not converge and two or more directions reached a sill, the directional maximum is returned in its place (`anisotropy_used = TRUE`), and a log warning names the directional ranges when their ratio exceeds 1.5. When the all-pairs estimate is used and the directional ranges vary by more than 1.5, a log note (INFO) names them instead.

The returned range is in the coordinate units of the (projected) data and can be passed directly to `make_folds(block_size = ...)` so 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 an unclassed 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; NA where that direction's fit was unusable), `anisotropy` (largest over smallest), `anisotropy_used` (logical: TRUE only when the all-pairs fit was singular or did not converge and the directional maximum stands in for it), `directional_status` (per azimuth, why a direction is NA in `directional`: "ok", "over_cutoff" (its range ran past the largest lag fitted), "not_converged" or "no_fit"),

`directional_fitted` (the range each direction's fit reported whether or not it was usable, so a refused directional range stays recoverable) and `directional_fits` (a list by azimuth of each direction's empirical variogram and fitted model, NULL where there is none, and NULL altogether unless `keep_directional_fits = TRUE`), `detrended` (logical: whether the variogram is of the residuals on `predictor_vars` or of 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), `detrend_method` ("ols" or "reml" when detrended, NA otherwise), `reml` (with `detrend = "reml"`: a list with `n_used`, `subsamped`, `nugget_prop` and `sigma2` from the REML fit; NULL otherwise), `crs` (the projected CRS the variogram was fitted in: the unit of the range), `max_dist`, `cutoff_dist`, `variogram` (the empirical variogram), `variogram_model` (the fitted gstat model, or with `detrend = "reml"` a gstat model built from the REML parameters) and `nugget` (that model's nugget variance; see [sac_nugget](#)), so the fit can be inspected and need not be taken on trust.

Rejected range `NA_real_` when a range was fitted but is not identified: it exceeds `range_frac * cutoff * max_dist` (see `range_frac`), which applies to the all-pairs fit even when some directions reached a sill; or too few pairs of points lie inside it to identify it, because it is shorter than the shortest lag the empirical variogram resolves (the mean separation in its first bin) or, with `detrend = "reml"`, than the distance within which 30 pairs of the points the REML fit used lie, when that is shorter (the REML range is fitted to the point pairs, not to the bins). A structure that short cannot be told from a nugget, and the bound is about identification, not a test for spatial structure (see above). Or the model did not converge; or the empirical variogram *decreases* with distance over its shorter lags (a net fall of more than 15 percent of the mean semivariance there, weighted by pairs), which is the shape of a periodic, hole-effect structure or of a variance that differs between a dense cluster and the rest of the layer. Sampling noise in the short-lag bins of a small sample can make that fall too: on exponential fields it refused 7–9 of 60 draws at `n = 30`, 3–6 at `n = 50` and 0–1 at `n = 100` (effective range 300 on a 1000 m square), and 15–16 of 60 at `n = 30` with a range of 150. An unremoved trend makes the variogram rise instead; when it rises past the fitted lags the first test catches it, but a milder trend only lengthens the fitted range and passes, which is what `predictor_vars` is for. Last, the fitted range can be non-positive. It is classed `sac_range` as well, so it prints as NA without dumping its attributes, and it carries `max_dist`, `cutoff_dist`, `variogram`, `variogram_model` and `nugget` (the evidence for the rejection), plus `rejected_range` (the value that was refused), `rejected_reason` (one of "fitted range exceeds the largest lag fitted", "fitted range is below the shortest lag fitted", "variogram model did not converge", "empirical variogram decreases with distance", "fitted range is non-positive or non-finite", "no variogram model could be fitted (singular fits)"), `crs` (so the units the rejected number was in stay recoverable, which is what `plot()` labels its axis from), `detrend_method`, `reml` (as on success) and, for "fitted range is below the shortest lag fitted", `range_floor` (the distance the refused range fell short of). The reason names one ground for the refusal: a falling variogram first, then a range past the largest lag, and the fit not having converged only when neither holds. A fit that did not converge is therefore most often recorded as past the largest lag, and the count of "variogram model did not converge" understates how many did not. The model itself says: `attr(attr(range, "variogram_model"), "converged")` is FALSE when no start of either model family converged (**gstat** stopped at its iteration limit), whatever the reason, and its nugget and sill are then where the optimiser halted, as its range is. A model built from REML parameters carries no such attribute. `resolution_profile()` reads it. The refused range carries `directional`, `anisotropy`, `anisotropy_used`, `directional_status`, `directional_fitted` and, with `keep_directional_fits`

= TRUE, directional_fits as well: the directional sweep runs whatever becomes of the all-pairs fit, and its per-azimuth outcome is what says whether any direction reached a sill the pooled variogram did not, or whether every direction ran past the fitted lags alike. The same shape, with rejected_range = NA, variogram_model = NULL and nugget = NA, is returned when no variogram model could be fitted at all (both the exponential and the spherical fit singular, which a flat, nugget-only variogram can produce, though on white noise it was the outcome in only 1 of 30 draws: see above); rejected_reason says so and the empirical variogram is still attached.

No fit An unclassed NA_real_ when estimation could not be attempted at all, whose one attribute, rejected_reason, says why: "package 'gstat', which the variogram needs, is not installed", "<n> points, fewer than the 30 a variogram range is estimated from", "<n> point(s) with a finite value to model, fewer than the 30 a variogram range is estimated from", "the response is constant", "the residuals on predictor_vars are constant: the predictors explain the response exactly" or "the points have no extent (the largest distance between them is zero or could not be computed)". Nothing is fitted, so none of the other attributes above exist: no variogram, variogram_model or rejected_range, which is what tells it from a range that was fitted and refused.

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.

Detrending and the residual-variogram bias

Fitting a variogram to the residuals of a least-squares trend underestimates both the sill and the range, because the trend fit absorbs part of the long-wavelength spatial variation (Lark, Cullis and Welham 2006). Blocks sized from that range are then too small and a blocked validation is less conservative than it claims. How large the effect is depends on how smooth the trend terms are in space, and on how many there are. Measured for this estimator on simulated exponential fields ($n = 300$, true effective range 300, nugget 0.2, 40–60 draws), as the median ratio of the estimate from the trend-removed data to the estimate from the true field:

- a white-noise covariate: OLS 1.00, REML 0.99 (no bias to speak of);
- a spatially smooth covariate (a random field with range 300 or 1000): OLS 0.97, REML 0.95–0.98;
- a linear trend in the coordinates: OLS 0.92, REML 1.04;
- a quadratic trend in the coordinates (five terms): OLS 0.75, REML 1.06.

So for ordinary covariates the OLS bias is a few percent, and for trend surfaces in the coordinates it is large. Iterating between a GLS trend fit and a variogram refit (Neuman and Jacobson 1984) recovers only part of it (0.80 in the quadratic case), because the variogram of GLS residuals is biased too; REML does not fit a variogram to residuals at all, which is why detrend = "reml" returns its own range estimate. Its price is a single family (exponential with nugget), a cubic cost in n , and a somewhat wider sampling spread. The default stays "ols" so that existing scripts return what they did; a script that detrends on smooth or coordinate-based terms should pass detrend = "reml", or size its blocks with a margin.

Count and other non-Gaussian responses

The empirical variogram assumes second-order stationarity: a variance that is the same everywhere, so that semivariance depends on separation alone. A count response breaks that assumption by

construction, because its variance tracks its mean, and so does any response with a mean-variance relationship (rates, proportions, skewed amounts). On such data the variogram mixes distance-dependence with mean-dependence, and the fitted range is not the thing it claims to be: where the mean is high the semivariance is inflated whatever the distance, which flattens the curve and can lengthen or shorten the apparent range depending on where the high-mean regions sit.

Supplying `predictor_vars` to `detrend` helps, because it removes the part of the mean the covariates explain, but it does not fix the variance structure: the residuals of an OLS fit to a count still have a variance that tracks the fitted mean. The principled remedy is a variogram of Pearson residuals from a model in the right family, which this function does not compute. Until it does, treat the range from a count response as no more than an order of magnitude, size blocks conservatively from it, and prefer `make_folds` (`method = "nndm"`), which does not depend on a fitted range at all.

References

- Lark, R. M., Cullis, B. R. and Welham, S. J. (2006). On spatial prediction of soil properties in the presence of a spatial trend: the empirical best linear unbiased predictor (E-BLUP) with REML. *European Journal of Soil Science*, 57(6), 787–799. doi:10.1111/j.13652389.2005.00768.x
- Neuman, S. P. and Jacobson, E. A. (1984). Analysis of nonintrinsic spatial variability by residual kriging with application to regional groundwater levels. *Mathematical Geology*, 16(5), 499–521. doi:10.1007/BF01886329

See Also

`sac_nugget` for the nugget behind the estimate, `plot.sac_range` to see the variogram the estimate rests on.

Other cross-validation: `area_of_applicability()`, `cv_bayes()`, `cv_block_size_sweep()`, `cv_gwr()`, `cv_rf()`, `cv_spatial()`, `fold_separation()`, `gwr_model_selection()`, `make_folds()`, `sac_nugget()`, `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 = 5e5 + runif(n, 0, 1000), y = 5e6 + 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")
  # print() throughout, because only the last value of a braced block is
  # shown on its own, and every one of these is worth reading.
  print(r) # the effective range, in metres
  print(attr(r, "directional")) # the four directional ranges
  print(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")

print(r2)
attr(r2, "rejected_range")
}

```

evaluate_insample	<i>Compute in-sample (or out-of-sample) metrics for fitted spatial models</i>
-------------------	---

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

<code>fits</code>	A <code>spatial_fit</code> object, or a named list of them (e.g. <code>list(GWR = gwr_obj, Bayesian = bayes_obj)</code>). 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. <code>model</code> is the key the comparison table is assembled on, so two fits sharing a name cannot be told apart in the output.
<code>newdata</code>	Optional <code>sf</code> object for out-of-sample evaluation. Must contain the response variable and all predictors. If <code>NULL</code> , in-sample metrics are computed.
<code>...</code>	Extra arguments passed to <code>predict()</code> .

Value

A `data.frame` with one row per model and columns for model name, all regression metrics, and `metric_basis`: what the row's metrics were computed on, "in-sample" (fitted values), "out-of-bag" (an `rf_fit`'s fitted values, see "What the metrics are computed on") or "newdata". Rows with different bases do not compare like for like. An element that is not a `spatial_fit` is skipped, with a logged warning, and has no row; a list in which no element is a `spatial_fit` is an error.

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 `data.frame` `model_metrics()` returns carries no label distinguishing the two, so check `object$info$fitted_are_oob` before comparing numbers across backends; `evaluate_insample()` and `compare_models()` record it per model in a `metric_basis` column. `compare_models_cv` scores every backend the same way.

R^2 is $1 - RSS/TSS$ with the total sum of squares taken about the mean of the response the model was *fitted* to. In sample that is the ordinary R^2 . With newdata it is out-of-sample R^2 , the convention every `cv_*()` function uses: the model is measured against the prediction it had to beat, the training mean, not against the new rows' own mean, which it could not have known. It is below 0 when the model predicts the new rows worse than the training mean does, and it is NA when the response does not vary about that baseline by more than rounding error (100 machine epsilons of its magnitude, whatever its units).

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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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()`, `model_metrics()`, `residual_morans_i()`

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = 5e5 + runif(60, 0, 1000), y = 5e6 + runif(60, 0, 1000),
              a = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  pts$z <- 2 * pts$a + rnorm(60, 0, 0.3)
  # Fit on 40 rows and keep 20 back, so the second call really is out of
  # sample; scoring the training rows again would only re-read the fit.
  fit <- fit_rf_model(pts[1:40, ], "z", "a", num_trees = 50, seed = 1)
  print(evaluate_insample(fit))           # in-sample (out-of-bag for RF)
  evaluate_insample(fit, newdata = pts[41:60, ]) # on the 20 held-out rows
}
```

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, and so kept out of 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,
  pointsize = "auto",
  boundary = NULL,
  .already_prepped = FALSE
)
```

Arguments

<code>data_sf</code>	An sf object with response, predictors, and geometries.
<code>response_var</code>	Response column name.
<code>predictor_vars</code>	Predictor column names. May be <code>character(0)</code> 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 and how much covariate effect (and the natural null model for comparing against a covariate model with `compare_models`).

family	A model family accepted by <code>brms::brm()</code> : a stats family function such as <code>poisson()</code> , or a brms family object such as <code>brms::zero_inflated_poisson()</code> , <code>brms::negbinomial()</code> , <code>brms::hurdle_poisson()</code> , <code>brms::bernoulli()</code> or <code>brms::Beta()</code> . Default NULL, resolved to <code>stats::gaussian()</code> . The family reaches <code>brms::brm()</code> unchanged with the spatial GP term still in the formula. A factor response is accepted only under <code>brms::categorical()</code> or an ordinal family, and those fits have no single expected value per row, so several methods cannot use them; see the section on non-Gaussian responses for what each family supports, and the count example below.
gp_k	Positive integer giving the number of GP basis functions <i>per dimension</i> , or NULL (default) to derive it from the length-scale/domain ratio. The fitted model carries gp_k^2 basis functions, not <code>gp_k</code> (see Details).
gp_c	Positive numeric boundary factor for the approximate GP, or NULL (default) to derive it alongside <code>gp_k</code> . 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. When you set <code>gp_c</code> and leave <code>gp_k = NULL</code> , <code>gp_k</code> is derived for <i>your</i> boundary: a wider boundary needs more basis functions to resolve the same length-scale, so raising <code>gp_c</code> raises the derived <code>gp_k</code> with it, up to the cap of 50 per dimension (a capped value is logged).
gp_iso	Logical; passed to <code>brms::gp(iso =)</code> . 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 makes the kernel anisotropic in the original CRS by whatever ratio $sd(X)/sd(Y)$ happens to take, because the coordinates are standardised per axis beforehand. See Details.
prior	Optional brms prior specification. When NULL and <code>standardize_predictors = TRUE</code> , weakly informative $\text{normal}(0, 5)$ priors are set on regression coefficients. A data-informed GP length-scale prior is always appended automatically unless prior already contains an entry with <code>class = "lscale"</code> .
chains	Number of MCMC chains. Default 4.
iter	Total iterations per chain. Default 2000.
warmup	Warmup iterations. Default $\text{floor}(\text{iter}/2)$.
cores	Number of cores for the sampler, one chain per core. Default <code>getOption("mc.cores", 1L)</code> , the same convention brms uses itself, so <code>options(mc.cores = 4)</code> once per session runs the four default chains in parallel everywhere. The previous default of <code>detectCores() - 1</code> 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, and rstan otherwise, which brms always brings. The cmdstanr package is a thin interface and can be installed without a CmdStan build (<code>cmdstanr::install_cmdstan()</code> builds one). An explicit

	"cmdstanr" with no usable CmdStan raises an error that says how to install it, so the failure does not come from inside the sampler.
control	Named list of sampler controls, <i>merged</i> over the package defaults <code>list(adapt_delta = 0.9, max_treedepth = 12)</code> instead of replacing them. Passing <code>list(max_treedepth = 15)</code> therefore keeps <code>adapt_delta = 0.9</code> , 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 (<code>\$info\$predictor_scaling</code> , a center and scale per predictor) so predictions can be computed correctly. The model is then fitted on the standardised predictors, so <code>coef()</code> reports a slope per standard deviation of each predictor and an intercept at the predictor means, not the raw-unit values <code>stats::lm()</code> would give; see coef.bayesian_fit .
check_convergence	Logical; after fitting, check for divergent transitions, R-hat above 1.05 and an effective-sample-size ratio below 0.1. Each problem found is written to the log as a WARN line (shown on the console unless spatialkit_quiet() is on), sets <code>\$info\$convergence_ok</code> to FALSE, and is detailed in <code>\$info\$convergence_diagnostics</code> ; <code>print()</code> on the fit flags it. The GP basis is also checked against the posterior length-scale (see Details): a basis too coarse for it is logged as a WARN line, and the share of draws it cannot resolve is recorded as <code>\$info\$convergence_diagnostics\$gp_1scale_below</code> but it does not change <code>convergence_ok</code> and <code>print()</code> does not flag it. None of these are raised as R warnings by the fit itself; the functions that score fits do raise one: cv_bayes() names the folds whose sampler did not converge (and marks them in <code>fold_metrics\$convergence_ok</code>), and compare_models() names such a model (column <code>convergence_ok</code>). Under rstan the sampler raises its own R-hat and ESS warnings; under cmdstanr nothing does, so read <code>\$info\$convergence_ok</code> . FALSE skips the checks and leaves <code>convergence_ok</code> NA (not checked). 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 <code>prep_model_data()</code> 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): `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`); when more than 10% of the posterior mass falls below it a WARN line is logged and the share is recorded as `$info$convergence_diagnostics$gp_1scale_b` which is the signal that `gp_k` should be raised. This runs with the other checks, so only under `check_convergence = TRUE`.

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.

`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_cmeans`: the column means `brms` centred the scaled coordinates on; `gp_xy_range`: the

training extrema of the scaled coordinates, which `predict()` uses, with `gp_S` and `gp_cmeans`, to hold the GP boundary at its fitted value; `gp_lengthscales_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` (TRUE or FALSE, and NA when nothing was checked, as under `check_convergence = FALSE`), `convergence_diagnostics`: `n_divergent`, `max_rhat`, `min_neff_ratio`, and `rhat_failed/neff_failed`, the parameters that failed each check by name with their values (empty when none failed), which is what makes a failed check actionable; and `n_dropped`: the rows `prep_model_data()` removed for missing or non-finite values or a bad geometry, so `$n` can be read against `nrow(data_sf)`). The raw `brmsfit` is in `$engine`.

Non-Gaussian responses

Nothing in this function is Gaussian-specific except its default. A numeric count, binary (0/1 or logical) or bounded response passes straight through to `brms` under the family you name; zero-inflated and hurdle counts, negative binomial, Bernoulli, beta, ordinal, categorical and mixture families have all been verified to reach `brms::brm()` with the GP term intact, the length-scale prior attached to each distributional parameter's GP.

The response check is family-aware. A factor or character response is accepted only under `brms::categorical()` or an ordinal family (`cumulative`, `sratio`, `cratio`, `acat`), and refused under every other family before anything is compiled; under gaussian a logical response is refused too. The case to watch is a two-level factor under `brms::bernoulli()`: `brms` would fit it, but `residuals()`, `summary()`, `model_metrics()` and `cv_bayes()` could not score a factor, so convert it to 0/1 first.

An ordinal or categorical fit has a probability per response category, not one expected value per row. `predict()` with its default `type = "epred"`, `fitted()`, `residuals()`, `summary()` and `model_metrics()` therefore stop with a message saying so, and `cv_bayes()` refuses the family before fitting anything. `predict(type = "predict", draws = TRUE)` returns the posterior predicted categories, as category indices, for new rows as well as the training ones (the share of draws in each category estimates its probability), and `brms::posterior_epred(fit$engine)` the probabilities for the training rows.

For the numeric families two things follow. First, the metrics that come back from `model_metrics()` and the `cv_*()` functions are not all meaningful for such a response: RMSE and MAE are, MAPE, SMAPE and R-squared are Gaussian-shaped, and for this backend `cv_bayes()`'s CRPS and interval coverage are the proper scores to read. See `model_metrics()`, section "Which metrics survive a non-Gaussian response". Second, `estimate_sac_range()` fits a variogram to the raw response (or OLS residuals), whose variance tracks its mean for a count; the range it reports is then less trustworthy than for a Gaussian response, and its help page says how.

One trap. The response check reads the family's name through `brms`'s own accessor; a family object it cannot name is treated as unknown and the check is skipped entirely, without falling back to either rule above. A malformed family therefore buys less validation, not more, and a wrong response type will surface as a Stan error, with no message from this function.

Spatial confounding

A fixed-effect coefficient estimated alongside a spatial random effect is a different quantity from the same coefficient in a non-spatial fit. When a covariate is itself spatially smooth, the GP term absorbs part of its effect and the two estimates can disagree sharply; when the response is smoother than the covariate, the estimate can shrink toward zero regardless of the true effect (Bolin and

Wallin 2025). Under a correctly specified spatial model this is not a bias but a change of estimand (Zimmerman and Ver Hoef 2022): the spatial coefficient is the effect *net of* whatever the spatial field can explain, and the non-spatial one is not. Which of the two a user wants depends on the question, so the honest diagnostic is to report both side by side and leave them unadjusted: fit the same formula with `stats::lm()` or `stats::glm()` and compare. Compare like with like: under `standardize_predictors = TRUE` the coefficients here are per standard deviation of each predictor, so either fit the non-spatial model on the same standardised columns or divide these slopes by `$info$predictor_scaling[[name]]$scale` first (see `coef.bayesian_fit`).

The literature on remedies is unsettled and this function takes no side. Restricted spatial regression (Hughes and Haran 2013) projects the spatial effect off the covariate space; later work argues it does not deliver what it promises (Hanks et al. 2015; Khan and Calder 2022). Within this backend, modelling the covariate and the spatial field as explicitly correlated (Marques, Kneib and Klein 2022) is expressible in a brms formula with no new code, and a spectral adjustment (Guan et al. 2023) is the principled route a Hilbert-space basis is well placed to support later.

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
- Bolin, D. and Wallin, J. (2025). Spatial self-confounding: smoothness-related estimation bias in spatial regression models. *Biometrika* **113**. doi:10.1093/biomet/asaf076
- Guan, Y., Page, G. L., Reich, B. J., Ventrucci, M. and Yang, S. (2023). Spectral adjustment for spatial confounding. *Biometrika* **110**, 699–719. doi:10.1093/biomet/asac069
- Hanks, E. M., Schliep, E. M., Hooten, M. B. and Hoeting, J. A. (2015). Restricted spatial regression in practice: geostatistical models, confounding, and robustness under model misspecification. *Environmetrics* **26**, 243–254. doi:10.1002/env.2331
- Hughes, J. and Haran, M. (2013). Dimension reduction and alleviation of confounding for spatial generalized linear mixed models. *Journal of the Royal Statistical Society: Series B* **75**, 139–159. doi:10.1111/j.14679868.2012.01041.x
- Khan, K. and Calder, C. A. (2022). Restricted spatial regression methods: implications for inference. *Journal of the American Statistical Association* **117**, 482–494. doi:10.1080/01621459.2020.1788949
- Marques, I., Kneib, T. and Klein, N. (2022). Mitigating spatial confounding by explicitly correlating Gaussian random fields. *Environmetrics* **33**, e2727. doi:10.1002/env.2727
- Zimmerman, D. L. and Ver Hoef, J. M. (2022). On deconfounding spatial confounding in linear models. *The American Statistician* **76**, 159–167. doi:10.1080/00031305.2021.1946149

See Also

Other model fitting: `fit_gwr_model()`, `fit_rf_model()`, `gp_lengthscales_bounds()`, `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 = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * (st_coordinates(dat)[, 1] - 5e5) +
    2 * dat$elev + rnorm(n)
  # Two short chains keep this to a few minutes; expect Stan to warn about
  # effective sample size, which is the cost of that. Use its defaults
  # (chains = 4, iter = 2000) for a fit to report.
  fit <- fit_bayesian_spatial_model(dat, "price", "elev",
                                   chains = 2, iter = 1000,
                                   compute_loo = FALSE)

  print(summary(fit))
  head(predict(fit, newdata = dat))

  # A zero-inflated count. The family is the only thing that changes.
  # summary()'s R2 and MAPE assume a Gaussian response, so for a count
  # read the coefficients here and score the model with cv_bayes(), whose
  # CRPS and interval coverage are defined for any response.
  dat$count <- rpois(n, exp(0.5 + 0.8 * dat$elev)) * rbinom(n, 1, 0.7)
  fit_zip <- fit_bayesian_spatial_model(dat, "count", "elev",
                                       family = brms::zero_inflated_poisson(),
                                       chains = 2, iter = 1000,
                                       compute_loo = FALSE)

  coef(fit_zip)
}

## 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

<code>data_sf</code>	An sf object with response, predictors, and geometries.
<code>response_var</code>	Response column name.
<code>predictor_vars</code>	Predictor column names (numeric columns; a name given twice counts once).
<code>adaptive</code>	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.
<code>bandwidth</code>	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. <code>prep_model_data()</code> 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 <code>sf::st_crs(fit\$data_sf)</code> , or pass <code>target_crs</code> to ensure_projected beforehand to fix the units yourself. If NULL (default), bandwidth is selected automatically via <code>GWmodel::bw.gwr()</code> . With <code>adaptive = FALSE</code> , 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. With <code>adaptive = TRUE</code> the count is rounded, and one too small for the model is raised, with a warning, to the smallest that gives every local regression more points of non-zero weight than parameters: the number of predictors plus 3 for the bisquare and tricube kernels (which give the farthest neighbour in a window weight 0), plus 2 for the others. That floor is enough unless several neighbours tie at the kernel's edge (a regular grid), which leaves a window fewer weighted points; then use a larger bandwidth. A count above the number of observations is capped at it, with a warning (a distance meant for <code>adaptive = FALSE</code> , most often). <code>bw.gwr()</code> searches adaptive bandwidths from 20 neighbours up, so below 20 observations its choice is capped the same way, with a warning, and is not an optimised bandwidth.
<code>kernel</code>	Kernel function type. One of "bisquare" (default), "gaussian", "tricube", "boxcar", "exponential".
<code>.already_prepped</code>	Logical (internal). If TRUE, skip the <code>prep_model_data()</code> 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` (NA, with a warning, where `GWmodel`'s `AICc` is undefined: its effective number of parameters $tr(S)$ is not below $n - 2$, the local regressions all but interpolate the data, and the large negative value `GWmodel` reports would rank the fit above any other), `bandwidth_is_fallback`

(TRUE when automatic selection failed and the arbitrary fallback was used), `condition_index` (the global index), `local_collinearity` (one row per observation: row, x, y, `n_window`, `cn` and `cn_slopes`; see **Collinearity diagnostics**), `n_local_collinear` (the locations whose slopes count as collinear), `n_local_singular` (the locations whose local coefficients came back non-finite), `nonfinite_coef` (a logical matrix, one row per observation and one column per term, with Intercept first, TRUE where the local coefficient came back non-finite, so the count in `n_local_singular` can be placed) and `n_dropped` (the rows `prep_model_data()` removed for missing or non-finite values or a bad geometry, so `$n` can be read against `nrow(data_sf)`). The raw GWmodel result is in `$engine`.

Collinearity diagnostics

The function computes **scaled condition indices** of the design and warns when one exceeds 30, the conventional threshold, which Wheeler & Tiefelsdorf (2005) carry over to the local designs of GWR. An index is the ratio of the largest to the smallest singular value (from an SVD) after each column is scaled (Belsley, Kuh & Welsch 1980); an exactly singular design gives Inf, which counts as the worst case, not an exempt one. 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 predictors centred at their means (with the intercept, which centring makes orthogonal to them), and kept as `info$condition_index`. It measures how nearly the predictors are collinear with one another over the whole study area; it is 1 for a single predictor, and a change of origin (degrees C or kelvin, a year or years since 2000) does not move it.

Local indices are then computed at **every** location, on the design the local regression there inverts: each row weighted by the square root of its kernel weight at the bandwidth the model is fitted with (supplied or selected), with rows of negligible weight dropped. A window left with fewer rows than columns counts as singular. Two indices are kept for each window, as columns of `info$local_collinearity`:

`cn` Belsley's index of the intercept plus the predictors, scaled to unit length but not centred. A predictor whose values in the window are far from 0 against their spread (a year, a temperature in kelvin) is collinear with the intercept and raises it: the local intercept is then an extrapolation to 0 and is ill-determined, but the slopes are not. It is what GWmodel's own solve sees.

`cn_slopes` The index for the slopes: the predictors centred at their weighted mean in the window and each divided by its standard deviation over the whole study area. It is 1 when the predictors vary as much, and as independently, inside the window as they do across the study area; it grows as a predictor becomes nearly constant inside the window (a regional covariate) or two predictors move together there. It does not depend on the predictors' origin or units.

A window's slopes count as collinear when `cn_slopes` is above 30 or singular, or when `cn` is above `1e6`, where GWmodel's uncentred solve starts to lose precision in the slopes too. Those windows are counted in `info$n_local_collinear`. A predictor that is constant, or nearly so, inside a window is caught this way whether it is alone or has company, so a single predictor is surveyed too.

A warning is issued whenever **any** location has collinear slopes; the wording reports a percentage when more than 25% of locations are affected and a count otherwise. Both are real R warnings, not log lines. Coefficients at a near-singular window are unstable and can be implausibly large. An **exactly** singular window (an indicator that is constant inside it, or fewer observations than parameters) makes GWmodel stop, so the fit fails with an error that says so; the window is not returned as NaN. A window with only `cn` above 30 raises no warning: `plot(fit, type = "coefficients"`,

term = "Intercept") masks it, and slope maps do not. Centre such a predictor if you want an interpretable local intercept.

After the fit, the local coefficient surfaces are scanned and a further warning counts local regressions that came back non-finite. GWmodel returns those where the kernel weights are undefined: with an adaptive bandwidth of k , a location where k or more observations share the same coordinates has a kernel of zero width, and every kernel but the boxcar divides 0 by 0 there. The warning names that cause when it applies. `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.

See Also

Other model fitting: `fit_bayesian_spatial_model()`, `fit_rf_model()`, `gp_lengthscales_bounds()`, `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 = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
               elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$price <- 10 + 0.01 * (st_coordinates(dat)[, 1] - 5e5) +
    2 * dat$elev + rnorm(n)
  fit <- fit_gwr_model(dat, "price", "elev", bandwidth = 30)
  print(summary(fit))      # print(): only a block's last value shows on its own
  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,
replace = TRUE,
sample_fraction = NULL,
seed = 123L,
num_threads = NULL,
.already_prepped = FALSE,
...
)

```

Arguments

<code>data_sf</code>	An sf object with response, predictors and geometry.
<code>response_var</code>	Response column name.
<code>predictor_vars</code>	Predictor column names.
<code>num_trees</code>	Number of trees. Default 500.
<code>mtry</code>	Predictors sampled per split. NULL uses ranger's default.
<code>min_node_size</code>	Minimum node size. NULL uses ranger's default.
<code>importance</code>	"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.
<code>include_coords</code>	Add the coordinates as predictors. Default FALSE; see above.
<code>replace</code>	Logical; grow each tree on a bootstrap sample drawn <i>with</i> replacement (TRUE, ranger's default and this one) or on a subsample drawn without replacement (FALSE). Strobl et al. (2007) show that bootstrap sampling with replacement is itself a source of bias in variable importance toward predictors with many distinct values or categories, and recommend subsampling without replacement. The forest's predictions change slightly with the choice; the default is kept at ranger's so that an existing script fits the same forest, and the setting is recorded in \$info and printed with the fit.
<code>sample_fraction</code>	Fraction of rows drawn for each tree. NULL (default) uses ranger's rule: all rows when <code>replace = TRUE</code> , 0.632 (the expected share of distinct rows in a bootstrap sample) when <code>replace = FALSE</code> . A single number in (0, 1] overrides it. <code>replace = FALSE</code> with <code>sample_fraction = 1</code> grows every tree on every row, so nothing is out of bag: the fit warns, and see What fitted() returns .
<code>seed</code>	Seed passed to ranger. Default 123.
<code>num_threads</code>	Threads for ranger. Default NULL means <code>getOption("mc.cores", 1L)</code> : 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 <code>cv_rf(parallel =)</code> would multiply it by the worker count). Predictions do not depend on the thread count, only speed does; pass <code>parallel::detectCores()</code> 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`, `replace`, `sample.fraction`, `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`). `$info` carries `num_trees`, `mtry`, `min_node_size`, `importance_type`, `importance` (a named numeric, or `NULL` when `importance = "none"`), `include_coords`, `replace` and `sample_fraction` (the sampling each tree was grown on, with `sample_fraction` resolved to the number `ranger` used), `oob_rmse` and `oob_r_squared` (each `NA_real_` when `ranger` did not compute it; forwarding `oob.error = FALSE` through `...` is one way to get there), `fitted_are_oob` (always `TRUE`; `summary()` reads it to label its metrics), `seed` and `n_dropped` (the rows `prep_model_data()` removed for missing or non-finite values or a bad geometry, so `$n` can be read against `nrow(data_sf)`). 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. That 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, once per session (it is a deliberate choice, so it is not raised as an R warning, and it is not repeated for every fold of a cross-validation). 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.

A row that every tree sampled has no out-of-bag prediction, and `ranger` reports `NaN` for it. That is every row under `replace = FALSE` with `sample_fraction = 1`, and a few under a small `num_trees`. The fit warns with the count; `fitted()` and `residuals()` are `NaN` on those rows, and `summary()`

says how many rows its metrics were computed on. With no row out of bag at all, the OOB error (NA in `$info$oob_rmse` and `$info$oob_r_squared`) and the permutation importance (NaN) are undefined too, and `print()` says so. `cv_rf()` scores its fold forests on the held-out rows, never out of bag, so it warns once with the number of folds affected rather than once per fold.

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)` when that importance is finite (it is NaN when no row is out of bag; see "What fitted() returns").

Other model fitting: `fit_bayesian_spatial_model()`, `fit_gwr_model()`, `gp_lengthscales_bounds()`, `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 = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              a = rnorm(n), b = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$z <- 2 * dat$a - dat$b + rnorm(n, 0, 0.3)
  fit <- fit_rf_model(dat, "z", c("a", "b"))
  print(fit)                # print(): only a block's last value shows on its own
  fit$info$importance       # permutation importance, a and b
}
```

fitted.bayesian_fit	<i>In-sample fitted values from a Bayesian spatial GP fit</i>
---------------------	---

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`. A posterior that cannot be drawn is an error, as is a family with a probability per response category (ordinal, categorical), which has no single fitted value per row.

The result is cached

`posterior_epred()` is $O(\text{draws} \times n)$, 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 it from scratch. 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, since the other simply recomputes), and `identical()` cannot distinguish two fits by their caches. The digest covers `data_sf` only. The entry is also tied to the engine that computed it – a refit or `update()` of the `brmsfit` is a different sampling run and recomputes – but a `brmsfit` edited by hand in place is what `clear_fitted_cache` is for. The entry holds only the values and a small identifier of the sampling run, so a fit saved with `saveRDS()` after `fitted()` is no larger for it.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

fitted.gwr_fit	<i>In-sample fitted values from a GWR fit</i>
----------------	---

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

<code>object</code>	A <code>gwr_fit</code> .
<code>...</code>	Ignored.

Value

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

See Also

Other methods on a fitted model: [coef.bayesian_fit\(\)](#), [coef.gwr_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.rf_fit\(\)](#), [model_metrics\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [print.spatial_fit\(\)](#), [residuals.bayesian_fit\(\)](#), [residuals.gwr_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

fitted.rf_fit	<i>Out-of-bag predictions from a random forest fit</i>
---------------	--

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 <code>rf_fit</code> .
...	Ignored.

Value

Numeric vector of length `object$n`.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

fold_separation	<i>How far the held-out points actually sit from the training data</i>
-----------------	--

Description

Blocked cross-validation exists to put distance between a test point and the training points that could tell you its value. Whether it succeeded is a measurement, and until now the package only offered a proxy for it: the block size compared against the estimated autocorrelation range, reported by `make_folds()` as a warning. That comparison is about the *design*. This is about the *result*: for every held-out point, the distance to its nearest training point, summarised per fold.

Usage

```
fold_separation(folds, data_sf, sac = NULL)
```

Arguments

folds	A <code>make_folds()</code> result, or its <code>\$folds</code> element (a list of train/test splits).
data_sf	The layer the folds were built on. Row identifiers are matched through <code>..row_id</code> when the layer carries one, and by row position otherwise, which is what <code>make_folds()</code> and every <code>cv_*</code> () do. As in <code>cv_*</code> (), a <code>make_folds()</code> result whose recorded rows sit at other locations here (folds built on another layer, such as the points before <code>assign_features_to_polygons()</code> dropped some) is refused; the location check is skipped when one of the two layers is POINT and the other is not.
sac	Optional: an <code>estimate_sac_range()</code> result or a single number. Defaults to the range the folds carry, if any. Supplying one adds the <code>within_range</code> column and the closing verdict. A range that records its CRS (the folds' own, or an <code>estimate_sac_range()</code> result) is compared with distances measured in that CRS, whatever CRS <code>data_sf</code> is in. A bare number is taken to be in the units the distances are otherwise measured in: those of <code>data_sf</code> if it is projected, and for geographic (lon/lat) input metres, in the CRS <code>ensure_projected()</code> chooses (as for <code>make_folds()</code> 's <code>block_size</code>), not degrees. A units object is refused.

Details

The two can disagree, and the direction is not obvious. Blocks wider than the range still leak wherever a test point sits near a block edge with training data just across it, which is most of the points in a fine block grid; conversely a fold whose blocks are narrower than the range can still separate well if the points inside them are clustered. The share of held-out points closer to training data than the correlation range is the number that settles it, and it is the last column here.

Nothing is estimated: the distances come from the geometry, and the range, when one is shown, is the one the folds already carry (`make_folds( auto_range = TRUE)` records it) or the one you pass as `sac`.

Value

A `data.frame` of class `fold_separation`, one row per fold: `fold` (the fold's number: for the `$folds` of a `cv_*`() result, the `fold_id` its `fold_metrics` use, which differs from the list position once a fold has been dropped), `n_train`, `n_test`, `n_blocks` (NA for a scheme with no blocks), `min_dist` and `median_dist` (distance from a held-out point to its nearest training point, in the units of the CRS the `crs` attribute names), and `within_range` (the share of held-out points closer to training data than `sac`; NA without one). Attributes: `method`, `sac_range`, `crs` (the CRS the distances were measured in) and `n_unknown_ids`.

See Also

[make_folds\(\)](#) for the fold schemes and the block sizing this measures the outcome of; [cv_block_size_sweep\(\)](#) for choosing a block size by cross-validated error instead.

Other cross-validation: [area_of_applicability\(\)](#), [cv_bayes\(\)](#), [cv_block_size_sweep\(\)](#), [cv_gwr\(\)](#), [cv_rf\(\)](#), [cv_spatial\(\)](#), [estimate_sac_range\(\)](#), [gwr_model_selection\(\)](#), [make_folds\(\)](#), [sac_nugget\(\)](#), [select_features_forward\(\)](#)

Examples

```
library(sf)
set.seed(1)
n <- 200
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000)),
  coords = c("x", "y"), crs = 32632
)

# Random folds put a training point almost on top of every held-out one.
random <- make_folds(pts, k = 4, method = "random_kfold", seed = 1)
print(fold_separation(random, pts, sac = 200))

# Blocked folds hold out whole neighbourhoods, so the distances grow.
blocked <- make_folds(pts, k = 4, method = "block_kfold",
  block_size = 250, seed = 1)
print(fold_separation(blocked, pts, sac = 200))
```

get_voronoi_seeds	<i>Generate seed points for Voronoi tessellation</i>
-------------------	--

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 only: 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. <code>n = nrow(sample_points)</code> is the common case, and yields <code>nrow(sample_points) - 1</code> seeds. Check <code>nrow()</code> on the result; do not assume n. Besides a number, n accepts what the level-selection step returned: the integer vector of ranked candidates from determine_optimal_levels() (its first element is used), a select_resolution() result (its <code>\$best</code>), or a resolution_profile() (read with <code>select_resolution()</code> at its default criterion). The result then carries <code>attr(, "n_from")</code> saying which. With method = "kmeans", a selection or profile also carries the centres of the partition the profile scored at that count, and those are returned instead of a new k-means: a k-means partition is the Voronoi partition of its centres, so their cells are the cells the criteria judged. <code>sample_points</code>, when given, is then only assigned to them (for <code>attr(, "kmeans")</code>), and <code>boundary</code> only sets the CRS; <code>kmeans_nstart</code>, <code>kmeans_iter</code> and <code>set_seed</code> are not used. Pass the count as a number (<code>n = sel\$best</code>) for a fresh k-means instead.

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, so they never reach <code>stats::kmeans()</code> , which fails on them without naming a cause. A lon/lat cloud, or one with no CRS whose coordinates look like lon/lat (the heuristic <code>ensure_projected()</code> applies, with its warning), is projected before clustering.
kmeans_nstart	Integer; nstart for <code>kmeans()</code> . Default 10. The partition is <code>stats::kmeans()</code> , not the best-of-25 k-means++ run that <code>resolution_profile()</code> scored a count on, so it is not that partition; pass the selection or profile itself as n for that one (see n).
kmeans_iter	Integer; iter.max for <code>kmeans()</code> . Default 100.
set_seed	Optional integer RNG seed.

Value

An sf POINT object with `seed_id` and `method` columns. With `method = "kmeans"` it also carries `attr(, "kmeans")`, the run behind the seeds: `cluster` (the `seed_id` each clustered cloud point was assigned to), `rows` (those points' positions in the cloud, since rows with unusable coordinates are dropped first), `size` (points per seed), `withinss` and `tot_withinss` (the within-cluster sums of squares), `iter` and `nstart`. For the scored centres of a selection or profile (see n) it describes `sample_points` assigned to them, each point to its nearest seed, with `iter` NA and `nstart` the profile's restarts, and is absent when no `sample_points` were given.

See Also

Other tessellation: `build_tessellation()`, `create_grid_polygons()`, `create_grid_polygons_cached()`, `create_voronoi_polygons()`, `ensure_stable_poly_id()`, `plot_tessellation_map()`, `voronoi_seeds_kmeans()`, `voronoi_seeds_random()`

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_lengthscales_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% is $\ell\sqrt{2\ln 20} \approx 2.45\ell$.

Usage

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

Arguments

<code>coords_xy</code>	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. <code>brms::gp()</code> defaults to <code>gr = TRUE</code> 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.
<code>q_small</code>	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.
<code>max_n</code>	Maximum number of <i>distinct</i> points to use in the distance computation. Default 1000. Set to <code>Inf</code> to use all of them.

Details

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

These are the bounds a length-scale *prior* is calibrated over, not the scales a fitted model can resolve: that depends on the basis size (`gp_k` in `fit_bayesian_spatial_model()`, which reports it as `$info$gp_ell_min`). Both bounds are fixed fractions of the spread of pairwise distances, so they do not shrink as points are added to the same area. A surface whose range sits below what the basis resolves needs a larger `gp_k`: more points help the data identify a short range, but they make neither these bounds nor the derived basis finer.

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.

See Also

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

Examples

```
set.seed(1)
xy <- cbind(runif(50), runif(50))
gp_lengthscales_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. `GWmodel` returns two loosely-coupled lists; this function returns a ranked table.

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

<code>data_sf</code>	An sf object with response, predictors and geometry.
<code>response_var</code>	Response column name.
<code>candidate_vars</code>	Character vector naming at least two numeric predictors to choose among. Factor, character and logical candidates are refused: <code>GWmodel</code> fits a factor as several model-matrix columns while this sweep counts it as one variable, so the criteria would not be comparable, and <code>fit_gwr_model()</code> , the documented next step, takes only numerics. Encode them as numeric indicators first.
<code>bandwidth</code>	Bandwidth held fixed across all candidate models. If <code>NULL</code> (default) it is selected with <code>GWmodel::bw.gwr()</code> on the model containing every candidate. Integer neighbour count when <code>adaptive = TRUE</code> ; otherwise a distance in the units of the projected CRS the sweep runs in, which <code>prep_model_data()</code> may have chosen for you. Geographic input is projected before the bandwidth is used, so a value in degrees would be read as metres; a fixed bandwidth below a ten-thousandth of the data's extent raises a warning saying so, as in <code>fit_gwr_model()</code> . An adaptive count too small for the full model is raised, with a warning, to the number of candidates plus 3 for the bisquare and tricube kernels (which give the farthest neighbour in a window weight 0), plus 2 for the others. That floor is enough unless several neighbours tie at the kernel's edge (a regular grid), which leaves a window fewer weighted points; then use a larger bandwidth. An adaptive count

	below 1 or above R's largest integer is refused, as in <code>fit_gwr_model()</code> . One above the number of observations is capped at it, with a warning; below 20 observations that includes <code>bw.gwr()</code> 's choice, since its adaptive search starts at 20 neighbours.
<code>adaptive</code>	Logical; adaptive (nearest-neighbour) bandwidth. Default TRUE.
<code>kernel</code>	Weighting kernel. One of "bisquare" (default), "gaussian", "tricube", "boxcar", "exponential".
<code>bw_approach</code>	Criterion for the bandwidth search: "AICc" (default) or "CV". Matches <code>fit_gwr_model</code> 's default.
<code>max_models</code>	Refuse the call if the sweep would exceed this many model fits. Default 200, which admits up to 19 candidates.
<code>dmat_max_n</code>	Precompute and reuse an $n \times n$ distance matrix when n is at most this. Default 2000 (about 32 MB). Set to 0 to disable.
<code>quiet</code>	Discard GWmodel's progress output. Default TRUE, because GWmodel writes it with bare <code>cat()</code> that no <code>suppressMessages()</code> 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.
<code>.engine</code>	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` is NA, and the model ranked last, where GWmodel could not evaluate it or where AICc is undefined because the model's effective number of parameters $tr(S)$ is not below $n - 2$, which raises a warning); `criterion` (label for the criterion actually read, noting when it had to be located positionally); `criterion_by_name` (logical: whether that column was found by its name rather than by the documented position), `criterion_column` (the column it was read from) and `criterion_verified` (logical: FALSE exactly when the column was read positionally from a table that did not have the four documented columns, which is the case the log calls unverified. Gate a script on this field instead of on the label); `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 follow from the method itself:

- **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-optimize 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 before it runs 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]])
```

Pass `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 Brunson, 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_block_size_sweep\(\)](#), [cv_gwr\(\)](#), [cv_rf\(\)](#), [cv_spatial\(\)](#), [estimate_sac_range\(\)](#), [fold_separation\(\)](#), [make_folds\(\)](#), [sac_nugget\(\)](#), [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 = 5e5 + runif(n, 0, 1000), y = 5e6 + 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$b + rnorm(n, 0, 0.5)
sel <- gwr_model_selection(dat, "z", c("a", "b", "noise"), bandwidth = 30)
print(sel)           # every model tried, by AICc
print(sel$best)      # the winning predictor set: a and b, not noise
fit <- fit_gwr_model(dat, "z", sel$best)
fit
}

```

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: anything <code>sf::st_crs()</code> accepts, including an sf or sfc object, whose CRS is used.
on_transform_error	What to do when <code>st_transform()</code> fails: "stop" (default) raises an error immediately; "set_crs" falls back to <code>st_set_crs()</code> (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.

See Also

Other spatial data preparation: [clip_target_for\(\)](#), [coerce_to_points\(\)](#), [ensure_projected\(\)](#), [prep_model_data\(\)](#)

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
c(a = st_crs(h$a)$epsg, b = st_crs(h$b)$epsg)
st_crs(h$a) == st_crs(h$b)
```

kriging_adequacy

Block-kriging adequacy diagnostics for a set of cells

Description

[summarize_by_cell\(\)](#) aggregates by plain means inside cell boundaries. A block-kriging aggregator would instead weight observations by their spatial correlation with the cell, and would return an estimate and a variance for every cell, thin or empty. Whether that is worth having on a given layer is a question with a measurable answer, and this function measures it, changing no cell value: for every cell it reports the block-kriging estimate and variance implied by a fitted variogram, that variance as a share of the variance the cell's mean would have with no data at all, and, where the cell has points, whether it exceeds the design-based variance of the plain mean, s^2/n ; and it scores the variogram itself by blocked cross-validation.

Usage

```
kriging_adequacy(
  assigned_points_sf,
  response_var,
  cells_sf,
  id_col = "poly_id",
  sac = NULL,
  folds = NULL,
  k = 5L,
  seed = 123L,
  nmax = 50L,
  max_neighbours = 2000L,
  max_box_ratio = 1000,
  quiet = TRUE
)
```

Arguments

<code>assigned_points_sf</code>	Points with a cell identifier column, as <code>assign_features_to_polygons()</code> returns.
<code>response_var</code>	The response column.
<code>cells_sf</code>	The cell polygons, with the matching ID column. A layer with no CRS is taken to be in the points' CRS (and points with none in the cells'), with a warning.
<code>id_col</code>	Preferred name of the ID column, found as <code>summarize_by_cell()</code> finds it: on the points the first of <code>id_col</code> , "poly_id", "polygon_id" and "cell_id"; on the cells the first of that column, "poly_id", "polygon_id", "id", "cell_id" and "grid_id". IDs are matched as text, whole numbers written out in full, so a double 1e5 matches an integer 100000; a point whose ID matches no cell is counted in no cell, with a warning. Default "poly_id".
<code>sac</code>	Optional <code>sac_range</code> carrying a variogram model.
<code>folds</code>	Optional <code>make_folds()</code> result on <code>assigned_points_sf</code> for the cross-validation statistic; built here with <code>block_kfold</code> when NULL. As in <code>cv_*</code> (), folds whose recorded rows sit at other locations in <code>assigned_points_sf</code> (built on another layer, such as the points before <code>assign_features_to_polygons()</code> dropped some) are refused.
<code>k, seed</code>	Folds and seed for that construction.
<code>nmax</code>	The number of neighbours each kriging system uses (<code>gstat</code> 's <code>nmax</code>): the locations nearest the cell's centre, or, for a cell those leave some of its own locations out of, all of its own plus this many outside it (see "The kriging neighbourhood"). The cross-validation kriges each held-out point from its <code>nmax</code> nearest training locations. Default 50.
<code>max_neighbours</code>	The largest kriging system a cell is given when its neighbourhood has to grow to hold all its own locations; a cell that would need more is left out (<code>kr_columns NA</code>) with a warning. Never below <code>nmax</code> . Default 2000.
<code>max_box_ratio</code>	A cell whose bounding box is more than this many times its area is left out (<code>kr_columns NA</code>) with a warning, because <code>gstat</code> 's discretisation of it costs memory in proportion. Default 1000.
<code>quiet</code>	Suppress progress messages. Default TRUE.

Value

An `sf` object of class "kriging_adequacy", one row per cell with the cell geometry and: the ID column, `n` (points in the cell), `mean` (the plain mean), `se` (its naive standard error), `kr_pred`, `kr_var`, `kr_ratio`, `kr_exceeds_design`, `kr_shift` and `kr_n_used` (how many locations the cell was kriged from; NA for a cell left out). Attributes: `variogram` (the model frame), `sill`, `nugget`, `range`, `range_identified`, `rejected_reason` (why the range was refused, from `sac`; NA when it was identified), `cv` (a list: `zscore_var`, `zscore_mean`, `rmse`, `n_pred`, `k`, `method`), `nmax`, `n_points` (the points used), `n_locations` (the distinct locations among them, which the kriging used) and `cells_left_out` (counts of cells left out, named shape for `max_box_ratio` and size for `max_neighbours`).

Reading the columns

kr_ratio The block-kriging variance over the cell's prior variance, in $[0, 1]$. The prior variance is the variance the cell's mean would have with no data at all, $\bar{C}(B, B)$: the covariance averaged over pairs of points in the cell, on the discretisation **gstat** block-kriges with, and without the nugget, which averages out over a block (**gstat** leaves it out of the block variance too). Each cell has its own: a cell's mean varies less than a single point does, and far less once the cell is wider than the range, so the point sill is not the scale. It is the coverage score, and it needs no hand-set threshold in metres or point counts: as it approaches 1 the estimate carries almost no information from the data about the cell and is reverting to the estimated mean. Ordinary kriging adds the variance of that estimated mean, so a cell the data do not reach comes out at or above its prior variance and reads 1. A cell at 0.05 is well determined; a cell at 0.8 is mostly prior. NA when the model is a pure nugget, where a cell mean has no prior variance to be a share of.

kr_exceeds_design TRUE where the kriging variance is larger than s^2/n from the cell's own points: kriging is not earning its keep there, and that is said per cell instead of globally. NA for cells with fewer than two points, where s^2 does not exist.

kr_shift The kriged estimate minus the plain mean, in units of the plain mean's standard error (NA where that is not defined). How much the aggregator would move the value, against the precision the value has.

The cross-validation statistic

The kriging variance is only as good as the variogram. With blocked folds, each held-out point is kriged from the training folds and the standardised error $(z - \hat{z})/\sqrt{kv}$ is recorded; the variance of those over all held-out points should be about 1. Its departure from 1 measures how badly the kriging variance is understated (or overstated): 1.4 means the variances are roughly 40 percent too small. Measured on simulated exponential fields ($n = 300$ on a 1000-unit extent, sill 1, nugget 0.2, five blocked folds, eight draws per configuration): 0.93–1.07 with the true variogram, 0.85–1.01 with the variogram estimated from the same points. With the nugget understated tenfold it moved only to 0.95–1.24. That is a property of the folds and not a weakness of the statistic: under blocked folds every held-out point is far from the training data, where the kriging variance is close to the sill whatever the nugget, so the blocked statistic checks the sill and range. To check the nugget, pass random folds (`make_folds(method = "random_kfold")`) as folds: the held-out points are then close to their neighbours, where the nugget decides the variance. The statistic is computed fold by fold with `gstat::krige()` on the splits `make_folds()` built, each held-out point kriged from that split's own training set, so the folds carry the same separation the package uses everywhere else: "buffered_loo" and "nndm" keep the points they exclude around each held-out one out of its kriging, and `print()` names the scheme that ran. A vector of fold labels is run as k-fold, each fold kriged from all the others.

What it said about block kriging as an aggregator

On the same simulated fields, with 16, 36 and 64 square cells: under uniform sampling the kriged and plain cell means differed by more than one standard error in 11–24 percent of cells and by more than two in 0–7 percent, the kriging variance exceeded s^2/n in 10–26 percent of populated cells, and at most one cell was empty. Under clustered sampling (eight clusters of 60-unit spread) the two aggregators parted: shifts above one standard error in 34–63 percent of cells and above two in 9–25 percent, the kriging variance below s^2/n in only 13–52 percent of populated cells, and 3–27

of the cells empty, each with a kriged estimate and variance where the plain mean has nothing. So a block-kriging aggregator earns its place on clustered layers and rarely on uniform ones, and this function says which kind a layer is.

What this needs

A variogram model. `sac` is a `estimate_sac_range()` result carrying one (`attr(, "variogram_model")`); when NULL it is estimated here from the response. The model families are the ones the package interprets elsewhere: exponential, spherical and Gaussian components with a nugget. Anything else is refused by name. A model whose range was not identified (an NA estimate with the model attached) is used with a warning that says why it was refused, and the reason is kept as `attr(, "rejected_reason")`: a range past the fitted lags means the sill was never reached and the ratios rest on an extrapolation; a fit that did not converge stopped wherever the optimiser halted; a variogram that falls with distance, or a range below the shortest lag, describes the data poorly at some lags.

The model has to be of the response itself. A variogram of residuals (`estimate_sac_range(predictor_vars = ...)`, `attr(, "detrended") TRUE`) leaves out the variance the predictors explain, while the response is kriged here without them, so `kr_var`, `kr_ratio` and the cross-validation statistic come out too small (the statistic at 4.3–5.2 against 0.67–1.53 with a spatially structured covariate); it is used with a warning. The points are put in the CRS the variogram was fitted in (`attr(sac, "crs")`), because its range is a length in that CRS's units. Requires **gstat**.

The kriging neighbourhood

gstat kriges a cell from the `nmax` locations nearest its centre. A cell holding more locations than that would be estimated from its middle alone, which describes the middle rather than the cell (in one simulated case a cell of 1,500 points came out 0.43 off at `nmax = 50`, against 0.07 from all of them and 0.02 for its plain mean). So wherever the `nmax` locations nearest a cell's centre leave out any of the cell's own locations, the cell is kriged from all of its own locations plus the `nmax` nearest outside it; `kr_n_used` says how many locations each cell was kriged from. The cost of a kriging system grows with the cube of its size (about 1 s at 2,000 locations and 17 s at 5,000), so a cell that would need more than `max_neighbours` is left out with a warning, its `kr_` columns NA.

gstat also discretises each cell into 500 points on a regular grid laid over the cell's whole bounding box, keeping those inside, so the memory a cell takes grows with the ratio of that box to its area: about 54 MB more for a thin diagonal strip at a ratio of 708, and 592 MB at 7,072. A cell whose bounding box exceeds its area more than `max_box_ratio` times (a sliver, parts far apart, a cell that is mostly hole) is left out the same way. `attr(, "cells_left_out")` counts both kinds, and `print()` says how many cells have no estimate and why.

Repeat measurements at one location

Two observations at the same coordinates (visits to a station, records geocoded to one address) make a kriging system singular, because **gstat** gives them the full sill, nugget included, as their covariance, as if they were one observation. The kriging and its cross-validation therefore use one observation per location, the mean of its replicates, with a warning; `n` and `mean` still count every point. How much of the nugget c_0 a mean of m replicates keeps is read off the replicates: their pooled within-location variance s_w^2 , capped at c_0 , is the part that differs from visit to visit and averages down, and the rest is micro-scale variation the visits share, so the mean carries error variance $c_0 - s_w^2 + s_w^2/m$.

That goes to **gstat** as a known measurement error (its weights) on the model with the nugget set to zero. For replicates that differ only by measurement error this is exactly the kriging of every observation, and for identical replicates it is the kriging of one; a location seen once is kriged as before. In the cross-validation a location is held out whole, under the fold of its first row, and its error variance is part of its standardised error. A cell or held-out location **gstat** still cannot krig and is reported NA with a warning, and `print()` says how many.

References

Cressie, N. (1993). *Statistics for Spatial Data*, revised edition. Wiley. (Block kriging, chapter 3; cross-validation of the kriging variance, section 2.6.4.)

See Also

Other aggregation: [assign_features_to_polygons\(\)](#), [determine_optimal_levels\(\)](#), [resolution_profile\(\)](#), [select_resolution\(\)](#), [summarize_by_cell\(\)](#), [summary.resolution_profile\(\)](#)

Examples

```
if (requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 200
  x <- 5e5 + runif(n, 0, 1000); y <- 5e6 + runif(n, 0, 1000)
  d <- as.matrix(dist(cbind(x, y)))
  z <- as.numeric(t(chol(0.8 * exp(-d / 100) + diag(0.2, n))) %*% rnorm(n))
  pts <- st_as_sf(data.frame(x = x, y = y, z = z), coords = c("x", "y"), crs = 32632)
  bnd <- st_sf(geometry = st_as_sfc(st_bbox(pts)))
  cells <- create_grid_polygons(bnd, target_cells = 16, type = "square")
  asg <- assign_features_to_polygons(pts, cells)
  ka <- kriging_adequacy(asg, "z", cells, k = 4)
  print(ka)
  # the report; print() because only a block's
  # last value shows on its own
  attr(ka, "cv")$zscore_var # about 1 when the variogram is right
}
```

Description

Builds train/test splits by random k-fold, spatial block k-fold, leave-location-out, buffered leave-one-out or nearest-neighbour distance matching (NNDM) leave-one-out.

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,
  blocks = NULL,
  balance_tol = 3
)
```

Arguments

- | | |
|-----------|--|
| points_sf | An sf object. Any Z or M dimension is dropped before folding: <code>sf::st_distance()</code> 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. They are re-projected when the coordinates look like lon/lat and otherwise stamped without reprojection, with a warning either way. |
| k | Integer; number of folds. Must be a single whole number ≥ 1 , and is required except for the two leave-one-out methods. 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. 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; "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. $k = 1$ is raised to 2 by "random_kfold", "block_kfold" and "leave_location_out", since one fold has no training set. Read the k element of the returned list, and do not assume the requested value. A reduction is written to the package log and raises no R warning, so <code>tryCatch(warning =)</code> will not see it and <code>suppressWarnings()</code> will not hide it. |
| 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, each a single whole number ≥ 1 . Give both, or give one and the other is derived from the extent's aspect ratio so that the blocks are roughly square. Ignored when block_size or auto_range override them.
block_multiplier	A single positive number, default 3. When neither block_size nor block_nx/block_ny is given, the automatic grid aims for $\text{block_multiplier} \times k$ blocks over the extent (aspect-preserving), so each fold holds out about block_multiplier blocks. An extent more than about $\text{block_multiplier} \times k$ times as wide as it is tall (or as tall as it is wide) gets a single row (or column) of that many 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. That is a CRS you did not choose, whose units are metres but whose identity varies with the data. block_size is then interpreted in <i>that</i> CRS. The CRS actually used is recorded in <code>params\$crs</code> 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.
auto_range	Logical. If TRUE, the spatial autocorrelation range is estimated via <code>estimate_sac_range()</code> (the omnidirectional, all-pairs range; the directional ranges it also fits are a diagnostic only, so on a field known to be anisotropic pass <code>block_size = max(attr(r, "directional_fitted"), na.rm = TRUE)</code> yourself after checking <code>attr(r, "directional_status")</code>, see estimate_sac_range) and used as the minimum block_size. Requires <code>response_var</code>. 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. blockCV takes the fitted variogram's range <i>parameter</i> as the block size, whereas this uses the <i>effective</i> range <code>estimate_sac_range()</code> returns (three times that parameter for an exponential fit), so its blocks are larger than blockCV's from the same variogram. When no range is identified, the geometric grid is used instead, with a warning that gives the reason. An identified range above half the extent in both directions leaves room for a single block, and <code>make_folds()</code> then stops with

	an error naming the size, rather than return one fold with an empty training set: pass a smaller <code>block_size</code> , or use <code>method = "nndm"</code> .
<code>range_frac</code>	Passed through to <code>estimate_sac_range()</code> when <code>auto_range = TRUE</code> . A fitted range beyond the longest lag the empirical variogram was fitted over is rejected as unidentified, and block sizing falls back to geometry (with a warning). A range within that bound can still be too wide for two blocks; see <code>auto_range</code> . Default 1.0.
<code>response_var</code>	Character(1) response column name. Required when <code>auto_range = TRUE</code> . For "block_kfold" a name that is not a column of <code>points_sf</code> is an error, whether or not <code>auto_range</code> is set: with it off the response still feeds the leakage warning, which a misspelt name would silently disable.
<code>group_var</code>	Character(1) naming a column of <code>points_sf</code> that identifies the location each observation belongs to. Required for <code>method = "leave_location_out"</code> , 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.
<code>prediction_points</code>	Optional sf layer of the locations you actually intend to predict onto. Required for <code>method = "nndm"</code> . The grid from <code>predict_surface()</code> 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. A point-to-polygon distance 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.
<code>predictor_vars</code>	Optional character vector of predictor column names. Passed to <code>estimate_sac_range()</code> for residual variogram estimation.
<code>boundary</code>	Optional polygonal sf/sfc for <code>block_kfold</code> . The grid is clipped to it; a cell that the boundary only touches at a corner or along an edge is not a block. A boundary without a CRS is brought into the points' CRS: reprojected from EPSG:4326 when its coordinates look like lon/lat, otherwise stamped, with a warning either way. The same goes for <code>prediction_points</code> and <code>blocks</code> .
<code>buffer</code>	For "buffered_loo": a single positive number, the distance within which the held-out point's neighbours are excluded from its training set. Like <code>block_size</code> it is in the units of the CRS the folds are built in (<code>params\$crs</code>), which for geographic (lon/lat) input is the metre CRS <code>ensure_projected()</code> chooses: 0.1 means 0.1 m, not 0.1 degrees. A buffer that excludes no neighbour from any fold makes the scheme plain leave-one-out, and is warned about.
<code>min_train</code>	For <code>method = "nndm"</code> : the smallest fraction of the data any fold's training set may be reduced to by neighbour exclusion. Default 0.5, as in <code>CAST::nndm()</code> . Where it binds, the distance matching stops short and the cross-validation stays optimistic; see Details .
<code>phi</code>	For <code>method = "nndm"</code> : the distance up to which the two nearest-neighbour distance distributions are matched, in the CRS the folds are built in. Matching is attempted only while a held-out point's nearest-neighbour distance is at most <code>phi</code> ; the exclusion that takes it past <code>phi</code> is the last one, so a realised distance can exceed <code>phi</code> by up to one neighbour step, as in <code>CAST::nndm()</code> . In Mila et

al. (2022), and in `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 $\phi = \text{"max"}$).

`drop_empty_blocks`

Logical. Default `TRUE`. With `FALSE` the blocks that hold no point are kept and packed into folds too, but `k` is still lowered to the number of blocks that hold points, so no fold is left without test points.

`blocks`

Optional polygon layer (`sf` or `sfc`, `POLYGON` or `MULTIPOLYGON`, at least two features) to use as the blocks of `"block_kfold"` in place of the grid this function would otherwise build: the `$cells` of a `build_tessellation()` result, hexagons, watersheds, administrative units, the `$blocks` of `blockCV::cv_spatial()`. Each point takes the block that contains it, and the blocks are then assigned to folds exactly as grid cells are. `block_size`, `block_nx/block_ny`, `block_multiplier` and `boundary` have nothing to act on and are ignored (logged); `auto_range` only compares the estimated range against the blocks. See **Supplied blocks** below for the CRS, overlap and coverage rules. An error for any other method.

`balance_tol`

A number of at least 1, default 3. For `"block_kfold"`: the ratio of the largest fold's point count to the smallest's above which the folds are reported as imbalanced, with a warning (an R condition, also logged) that names both counts. `Inf` disables the check. The value is the tolerance of a check, not a target the packing aims for: see **Fold balance** below for what the packing can and cannot do. `params$balance_ratio` carries the ratio achieved.

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): it sizes the exclusion around each held-out point, with no arbitrary buffer, 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, 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 is matched only while its nearest-neighbour distance is at most ϕ (default: the largest prediction distance, since a training point already further than every prediction distance has nothing to match), so the exclusion

that takes it past ϕ is its last and a realised distance can exceed ϕ by up to one neighbour step; and no fold's training set is stripped below `min_train` of the data.

It differs from `CAST::nndm()` in two details, so the folds agree closely with CAST's but not exactly. The removal rule is strict: a neighbour is removed whenever the realised distribution exceeds the target, whereas CAST removes one only while the realised distribution, less the point about to move, is still at or above the target. The rule here therefore removes up to one point more per distance value (a few percent more removals in all on clustered layouts), erring on the pessimistic side. And ties in G_j^* are broken by the points' coordinates, not by their row index as in CAST, so the folds do not depend on the order of the rows.

Where neither limit binds, 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. Beyond ϕ no matching is attempted, by design. `min_train` is different: when the prediction locations lie further from the samples than a fold can be made to hold out (clustered samples and a prediction domain well beyond them, the layout NNDM is meant for), it stops the matching early and the realised distances stay optimistic. `params$n_at_min_train` counts the folds that were held at the floor while still closer than the target allows, and `make_folds()` warns when that leaves more than one point's worth of excess at or below ϕ . Lower `min_train` to match further, or read the cross-validated score as an upper bound on performance at the prediction locations.

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% of folds had a nearest training point within 50 m against a target of 9%), an *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 "block_kfold" the block design is returned with the folds. `assignment` has a third column, `block_id`: the block each point fell in, numbered as the rows of `params$blocks`, an sf layer of the block polygons in the CRS the folds were built in. Those rows are **not** the rows of the layer the blocks came from: `drop_empty_blocks = TRUE` (the default) removes the blocks that hold no point and rennumbers the rest, so with supplied blocks a nine-polygon layer of which three are empty comes back as six rows numbered 1 to 6. `params$blocks$source_row` is the row each one came from in that original layer (a grid cell's index in the full `grid_nx` by `grid_ny` grid, or the row of the blocks argument), so `blocks[params$blocks$source_row, ]` recovers them with their own columns and in their own order. It runs from 1 to `params$n_blocks` and is the identity when nothing was dropped. For a grid `params$n_blocks` is `grid_nx * grid_ny`, the cells a boundary clips away included, and `params$blocks_used` is the number of blocks returned. `params$block_sizes` is the number of points in each block, indexed by `block_id` (zeros are empty blocks that `drop_empty_blocks = FALSE` kept), and `params$fold_blocks` is a list with one integer vector per fold naming the blocks packed into it. Between them the folds account for every block

exactly once, empty ones included, so a fold's territory on the map is all of its blocks and not merely the ones that happen to hold points. So `table(assignment$fold)` can be traced back to the blocks it is made of, a fold can be seen to be one contiguous region or several, and the blocks can be drawn over the data (`plot_folds()` does so).

For "block_kfold", `params` carries a `params$blocks_supplied` that says whether the blocks came from blocks or from a grid built here, and `params$boundary_supplied` whether a boundary was given. Every method's `params` carries `params$row_probe`, a small sample of row IDs and coordinates that every `cv_*`() compares against the data it is handed, so folds built from a different layer of the same size are refused, never applied silently.

For the methods that work in projected space, `params` also 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 for you, and not 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.

Supplied blocks

A polygon layer passed as blocks is aligned to the points the way boundary is: CRS-less points are aligned to blocks that carry a CRS (reprojected if they look like lon/lat, otherwise stamped, warning either way), and the blocks are then brought into the CRS the folds are built in. A point inside more than one block is given the first (lowest row) that contains it, as for a point on the shared edge of two grid cells. When the blocks that caught such a point share area instead of an edge, the layer overlaps and is not a partition, and this is warned about. A point inside no block is assigned to the nearest one, by distance to the polygon itself, and the count of such points is warned about, unless they sit within a millionth of the extent of a block, which is an edge that reprojection or clipping moved by a rounding error. Blocks that hold no point are dropped when `drop_empty_blocks = TRUE`, and `k` is lowered to the number of blocks that hold points when that is smaller. `params$n_blocks` is the number of blocks before empties were dropped, `params$blocks_used` the number after (so with `drop_empty_blocks = FALSE` the two are equal, and `sum(params$block_sizes > 0)` is how many of them hold points), `params$grid_nx` and `params$grid_ny` are NA, and `params$block_scale` is the median over blocks that hold points of the side of the square with the block's area. That is the length compared against the autocorrelation range for the leakage warning, since a polygon has no single edge length.

The connection to the rest of the package is `build_tessellation()`: every shape it builds can be a block design here, including Voronoi cells around `get_voronoi_seeds(method = "kmeans")` seeds, which adapt to the density of the points.

Fold balance

Only "block_kfold" balances the number of points per fold, and it does so by packing: blocks are taken largest first and each goes to the fold with the fewest points so far, ties broken at random. This is the longest-processing-time rule for multiway partitioning. Measured against the optimum by enumeration (two folds, up to ten blocks, heavy-tailed block sizes) it is optimal in 72 percent of cases and within two points of optimal on average; a local search with 30 random restarts moved the largest-to-smallest ratio by 0.004 on average and never brought a packing above the 3:1 tolerance below it. An imbalance past the tolerance is therefore in the points per block, which no assignment of blocks to folds can even out, and this function offers no search over packings. The remedy is

the block design: smaller blocks, or blocks that adapt to the density of the points passed through blocks. On clustered layouts where the geometric grid exceeded 3:1 in 22 percent of cases (median ratio 1.8, worst 6.3), Voronoi cells around 15 k-means seeds never exceeded 1.3 (median 1.09). Density-adaptive blocks are smaller where points are dense, so check `params$block_scale` against the autocorrelation range as you would a grid.

The other methods do not balance point counts. `"random_kfold"` is balanced by construction (fold sizes differ by at most one); `"buffered_loo"` and `"nndm"` hold out one point per fold; `"leave_location_out"` gives each fold the same number of *locations* (to within one), so folds differ by as much as the locations' sizes do.

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](https://doi.org/10.1111/2041210X.13851)
- Roberts, D. R., Bahn, V., Ciuti, S., Boyce, M. S., Elith, J., Guillera-Aroita, 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](https://doi.org/10.1111/ecog.02881)
- Valavi, R., Elith, J., Lahoz-Monfort, J. J. and Guillera-Aroita, 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](https://doi.org/10.1111/2041210X.13107)

See Also

Other cross-validation: [area_of_applicability\(\)](#), [cv_bayes\(\)](#), [cv_block_size_sweep\(\)](#), [cv_gwr\(\)](#), [cv_rf\(\)](#), [cv_spatial\(\)](#), [estimate_sac_range\(\)](#), [fold_separation\(\)](#), [gwr_model_selection\(\)](#), [sac_nugget\(\)](#), [select_features_forward\(\)](#)

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(30, 0, 1000), y = 5e6 + runif(30, 0, 1000)),
  coords = c("x", "y"), crs = 32632
)
folds <- make_folds(pts, k = 3, method = "block_kfold", seed = 42)
folds$assignment      # fold and block membership per row
lengths(folds$folds[[1]]) # train/test row-ID splits
folds$params$block_sizes # points per block; params$fold_blocks packs them

# 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

<code>object</code>	A <code>spatial_fit</code> object.
<code>...</code>	Additional arguments passed to <code>predict()</code> .
<code>newdata</code>	Optional <code>sf</code> object for out-of-sample evaluation. If <code>NULL</code> , fitted values are used (in-sample, or out-of-bag for an <code>rf_fit</code> ; 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`, `n_MAPE` and `n_SMAPE` (the last two are the rows each percentage error was averaged over; see "Percentage errors on responses with zeros"). `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 `data.frame` `model_metrics()` returns carries no label distinguishing the two, so check `object$info$fitted_are_oob` before comparing numbers across

backends; `evaluate_insample()` and `compare_models()` record it per model in a `metric_basis` column. `compare_models_cv` scores every backend the same way.

R^2 is $1 - RSS/TSS$ with the total sum of squares taken about the mean of the response the model was *fitted* to. In sample that is the ordinary R^2 . With newdata it is out-of-sample R^2 , the convention every `cv_*()` function uses: the model is measured against the prediction it had to beat, the training mean, not against the new rows' own mean, which it could not have known. It is below 0 when the model predicts the new rows worse than the training mean does, and it is NA when the response does not vary about that baseline by more than rounding error (100 machine epsilons of its magnitude, whatever its units).

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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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.

Which metrics survive a non-Gaussian response

RMSE and MAE are defined for any numeric response and are what to read for a count, a rate or a bounded outcome. MAPE and SMAPE assume a response that is rarely zero (see the previous section), and R-squared and adjusted R-squared compare residual variance to total variance, which is the right comparison for a Gaussian response and a loose one for anything whose variance tracks its mean. None of the four is wrong to compute; each is Gaussian-shaped thinking, and on a Poisson or zero-inflated response should be read as a rough summary rather than a score.

For the Bayesian backend, `cv_bayes()` additionally reports CRPS and interval coverage at 50, 80 and 95 percent. Both are proper scoring rules computed from posterior draws, so they are meaningful for any family that predicts one number per row (a count, a rate, a binary or bounded outcome), and they are the numbers to compare when the response is not Gaussian. A categorical or ordinal family predicts a probability per response category instead, so `cv_bayes()` refuses one before fitting anything. When every fold fails, the `fold_metrics` frame `cv_bayes()` returns carries the CRPS column but not the `coverage_*` columns, so code that reads those columns must tolerate their absence.

See Also

Other model evaluation: [compare_models\(\)](#), [compare_models_cv\(\)](#), [evaluate_insample\(\)](#), [residual_morans_i\(\)](#)

Other methods on a fitted model: [coef.bayesian_fit\(\)](#), [coef.gwr_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.gwr_fit\(\)](#), [fitted.rf_fit\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [print.spatial_fit\(\)](#), [residuals.bayesian_fit\(\)](#), [residuals.gwr_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

Examples

```
if (requireNamespace("ranger", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  pts <- st_as_sf(
    data.frame(x = 5e5 + runif(60, 0, 1000), y = 5e6 + runif(60, 0, 1000),
              a = rnorm(60)),
    coords = c("x", "y"), crs = 32632
  )
  pts$z <- 2 * pts$a + rnorm(60, 0, 0.3)
  # Fit on 40 rows and keep 20 back, so the second call really is out of
  # sample; scoring the training rows again would only re-read the fit.
  fit <- fit_rf_model(pts[1:40, ], "z", "a", num_trees = 50, seed = 1)
  print(model_metrics(fit))           # in-sample (out-of-bag for RF)
  model_metrics(fit, newdata = pts[41:60, ]) # on the 20 held-out rows
}
```

new_spatial_fit	<i>Build a spatial_fit S3 object</i>
-----------------	--------------------------------------

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: <code>predict()</code> , <code>fitted()</code> , <code>residuals()</code> and <code>coef()</code> on the result all dispatch on it, so a custom subclass requires a matching <code>predict.<subclass>()</code> method to be usable with <code>cv_spatial()</code> .
engine	The raw model object your backend produced (an <code>lm</code> , a <code>ranger</code> object, a <code>brmsfit</code> , ...). Nothing inspects it except your own methods.
formula	A formula.
response_var	Character(1).
predictor_vars	Character vector.
data_sf	An <code>sf</code> object used for fitting.
info	Named list of model-specific extras. Set <code>fitted_are_oob = TRUE</code> when your <code>fitted()</code> values are held out instead of in-sample, so <code>summary()</code> labels them correctly.

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 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()`, `gp_lengthscales_bounds()`, `prep_model_data()`

Examples

```
library(sf)
set.seed(1)
n <- 80
site <- st_as_sf(
  data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
             elev = rnorm(n)),
  coords = c("x", "y"), crs = 32632
)
site$price <- 10 + 0.01 * (st_coordinates(site)[, 1] - 5e5) +
  2 * site$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.aoa

Plot the dissimilarity distribution behind an area of applicability

Description

`n_outside` says how many prediction locations fall outside the area of applicability; it does not say whether the rest sit comfortably inside or crowd against the threshold, nor how far outside the outsiders are. This draws the dissimilarity index of the prediction locations against that of the training data, with the threshold marked, so the prediction set can be read as mostly inside, marginal or largely outside. The training DI is cross-validated over the folds passed to [area_of_applicability\(\)](#), or without them is each training point's distance to its nearest other training point; the legend and the caption say which. The training curve is the reference the threshold was derived from: the

threshold is the largest training DI inside an outlier fence, so the curve reaches it exactly when no training value was fenced off and runs past it, by the tail the fence removed, when some were. A prediction location outside on a predictor dropped for having no training variance has DI = Inf: it counts in the prediction curve, which then tops out below 1, and the caption says how many are off the axis. A location with a missing predictor (DI = NA) is neither inside nor outside; it is left out of the curve and the caption counts it.

Usage

```
## S3 method for class 'aoa'
plot(x, type = c("ecdf", "histogram"), ...)
```

Arguments

x	An aoa object from area_of_applicability() .
type	"ecdf" (default), the two empirical distribution functions on one axis, or "histogram", the prediction DI as bars with the training DI as an outline.
...	Ignored.

Value

A ggplot object.

See Also

Other plotting: [plot.block_size_sweep\(\)](#), [plot.feature_selection\(\)](#), [plot.gwr_model_selection\(\)](#), [plot.resolution_profile\(\)](#), [plot.sac_range\(\)](#), [plot.spatial_fit\(\)](#), [plot_calibration\(\)](#), [plot_cv_metrics\(\)](#), [plot_folds\(\)](#)

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(2)
  n <- 200
  train <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
               a = rnorm(n), b = rnorm(n)),
    coords = c("x", "y"), crs = 32632)
  train$z <- train$a - train$b + rnorm(n, 0, 0.3)
  # Prediction locations whose predictor `a` drifts beyond the training range.
  new <- st_as_sf(
    data.frame(x = 5e5 + runif(100, 0, 1000), y = 5e6 + runif(100, 0, 1000),
               a = rnorm(100, mean = 2), b = rnorm(100)),
    coords = c("x", "y"), crs = 32632)
  aoa <- area_of_applicability(new, train_sf = train, predictor_vars = c("a", "b"))
  print(plot(aoa))
  # the two ECDFs; print() because only a
  # block's last value is drawn on its own
  plot(aoa, type = "histogram")
}
```

plot.block_size_sweep *Plot cross-validation error against block size*

Description

The curve from `cv_block_size_sweep()`: the pooled metric at each block size, the fold-to-fold range as a band, the random-fold reference as a dashed line, and the estimated autocorrelation range as a vertical marker. Blocks smaller than the range leak, so the curve rises from the reference towards the range and plateaus beyond it; the height of the rise is what the random-fold number overstated. The caption says which way is better: higher for R2 and Adj_R2, closer to the nominal level for a coverage_* column (0.975 for coverage_97.5), since over-coverage is miscalibration too, and lower for everything else.

Usage

```
## S3 method for class 'block_size_sweep'
plot(x, ...)
```

Arguments

x	A block_size_sweep.
...	Ignored.

Value

A ggplot object.

See Also

Other plotting: `plot.aoa()`, `plot.feature_selection()`, `plot.gwr_model_selection()`, `plot.resolution_profile()`, `plot.sac_range()`, `plot.spatial_fit()`, `plot_calibration()`, `plot_cv_metrics()`, `plot_folds()`

Examples

```
if (requireNamespace("ranger", quietly = TRUE) &&
    requireNamespace("gstat", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 200
  x <- 5e5 + runif(n, 0, 1000); y <- 5e6 + runif(n, 0, 1000)
  d <- as.matrix(dist(cbind(x, y)))
  field <- as.numeric(t(chol(exp(-d / 100) + diag(1e-6, n))) %*% rnorm(n))
  dat <- st_as_sf(data.frame(x = x, y = y, a = rnorm(n)), coords = c("x", "y"),
                  crs = 32632)
  dat$z <- field + 0.5 * dat$a + rnorm(n, 0, 0.2)
  rf_fn <- function(train_sf)
    fit_rf_model(train_sf, "z", "a", include_coords = TRUE, num_trees = 100,
```

```

      seed = 1)
sw <- cv_block_size_sweep(dat, "z", "a", fit_fn = rf_fn, k = 4, n_sizes = 4,
                          quiet = TRUE)
# Error rises from the random-fold reference (dashed) as the blocks grow
# towards the estimated autocorrelation range (vertical marker). The
# height of that rise is what random folds were hiding; with four sizes the
# ladder stops near the range, so the plateau beyond it is not drawn here.
plot(sw)
}

```

plot.feature_selection

Plot the path of a forward feature selection

Description

[select_features_forward\(\)](#) scores every candidate at every step and keeps the best; its history holds all of them. This draws the accepted variable's score at each step as the path, every other candidate's score at that step as a faint point, and the step at which the selection stopped in red. When it stopped because no candidate cleared `tol`, the path runs one step further to the best of the rejected candidates, drawn hollow and labelled "not added", so the stop reads as a flattening rather than a cut. The picture then says whether the last variable was a clear gain or the first that happened to clear `tol`, and whether the runner-up would have done as well. The scores are the selection's own cross-validated criterion, optimistically biased by the selection (see the help page's section on that); when a hold-out score was computed (`select_on = "split"`) it is drawn as a separate mark at the final step and named in the caption.

Usage

```
## S3 method for class 'feature_selection'
plot(x, ...)
```

Arguments

<code>x</code>	The list returned by select_features_forward() .
<code>...</code>	Ignored.

Value

A ggplot object.

See Also

Other plotting: [plot.aoa\(\)](#), [plot.block_size_sweep\(\)](#), [plot.gwr_model_selection\(\)](#), [plot.resolution_profile\(\)](#), [plot.sac_range\(\)](#), [plot.spatial_fit\(\)](#), [plot_calibration\(\)](#), [plot_cv_metrics\(\)](#), [plot_folds\(\)](#)

Examples

```

if (requireNamespace("ranger", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(4)
  n <- 150
  dat <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              a = rnorm(n), b = rnorm(n), c = rnorm(n)),
    coords = c("x", "y"), crs = 32632)
  dat$z <- 2 * dat$a - dat$b + rnorm(n, 0, 0.5)
  fit_fn <- function(train_sf, vars)
    fit_rf_model(train_sf, "z", vars, num_trees = 80, seed = 1)
  sel <- select_features_forward(dat, "z", c("a", "b", "c"), fit_fn = fit_fn,
                                k = 3, quiet = TRUE)

  plot(sel)
}

```

plot.gwr_model_selection

Plot a GWR model selection

Description

`gwr_model_selection()` ranks every model it evaluated on AICc. This draws each model's criterion against its number of predictors, the winner in red, so the gap between the best model and the runners-up (which the ranked table shows only as numbers) is read as a shape: a winner well below the rest was chosen by the data, a winner a fraction of an AICc unit ahead of three others was chosen by the tie-break. The criterion is in-sample and the caption carries the label `gwr_model_selection()` attached to it, including any note saying it was read positionally.

Usage

```
## S3 method for class 'gwr_model_selection'
plot(x, ...)
```

Arguments

<code>x</code>	A <code>gwr_model_selection</code> object.
<code>...</code>	Ignored.

Value

A ggplot object.

See Also

Other plotting: `plot.aoa()`, `plot.block_size_sweep()`, `plot.feature_selection()`, `plot.resolution_profile()`, `plot.sac_range()`, `plot.spatial_fit()`, `plot_calibration()`, `plot_cv_metrics()`, `plot_folds()`

Examples

```
if (requireNamespace("GWmodel", quietly = TRUE) &&
    requireNamespace("sp", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 80
  dat <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + 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)
  # Every model tried, AICc against its size; the winner (a and b) marked.
  plot(sel)
}
```

plot.resolution_profile

Plot a resolution profile

Description

The criteria of a [resolution_profile\(\)](#) against the number of cells, one panel per criterion on a shared x axis, with the level each criterion selects marked and the region over which it is within `tol` of its optimum shaded. That shaded band is the flat region [select_resolution\(\)](#) reports. A criterion whose optimum sits at an end of the levels it was scored at is captioned with which bound that is (the support ceiling, the range floor, the ladder's own end, or the first level the criterion is computable at), because there the bound is choosing, not the criterion. A criterion that is NA at every level is left out with a log line rather than drawn empty.

Usage

```
## S3 method for class 'resolution_profile'
plot(x, criteria = NULL, tol = 0.02, ...)
```

Arguments

<code>x</code>	A <code>resolution_profile</code> .
<code>criteria</code>	Character vector of criteria to draw, any of "cp", "reliability", "elbow", "moran_z", "wss". Default: every one of the four selectable criteria that is finite at some level. "wss" is the raw curve behind "elbow" and has no selected point.
<code>tol</code>	Passed to select_resolution() for the flat region.
<code>...</code>	Ignored.

Value

A ggplot object.

See Also

Other plotting: [plot.aoa\(\)](#), [plot.block_size_sweep\(\)](#), [plot.feature_selection\(\)](#), [plot.gwr_model_selection\(\)](#), [plot.sac_range\(\)](#), [plot.spatial_fit\(\)](#), [plot_calibration\(\)](#), [plot_cv_metrics\(\)](#), [plot_folds\(\)](#)

Examples

```
if (requireNamespace("gstat", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  # The same field as ?resolution_profile: an exponential covariance with
  # range parameter 200 and a nugget of 0.6 on a unit sill.
  set.seed(4)
  n <- 400
  xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000))
  D <- as.matrix(dist(xy))
  xy$z <- as.numeric(t(chol(exp(-D / 200) + diag(0.6, n))) %*% rnorm(n))
  pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
  prof <- resolution_profile(pts, response_var = "z", n_levels = 12)
  print(plot(prof))
  # one panel per criterion it scored: no
  # elbow on these uniform points
  plot(prof, criteria = c("cp", "wss")) # Cp beside the raw WSS curve
}
```

plot.sac_range

Plot an estimated spatial autocorrelation range

Description

Draws the empirical variogram that [estimate_sac_range\(\)](#) attaches to its result, with the fitted model and the effective range overlaid where a range was identified, and a subtitle saying why not where it was not. A variogram that never reaches a sill, or that has no spatial structure at the lags resolved, is the single most useful thing to *see* when a range comes back NA, so those cases are drawn rather than refused.

Usage

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

Arguments

x An object of class `sac_range`, as returned by [estimate_sac_range\(\)](#).

... Ignored.

Details

Nothing is recomputed: the plot reads the variogram, variogram_model, crs, directional and anisotropy_used attributes the estimate already carries. The distance axis is in the units of the CRS the variogram was actually fitted in, which estimate_sac_range() may have chosen itself for lon/lat input.

Value

A ggplot object.

See Also

`plot.spatial_fit`(type = "variogram"), which draws the same picture for a fitted model's residuals.

Other plotting: `plot.aoa()`, `plot.block_size_sweep()`, `plot.feature_selection()`, `plot.gwr_model_selection()`, `plot.resolution_profile()`, `plot.spatial_fit()`, `plot_calibration()`, `plot_cv_metrics()`, `plot_folds()`

Examples

```
if (requireNamespace("gstat", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  # An exponential field with range parameter 150 (effective range about
  # 450 m) and a nugget of half the sill, so the fitted model, the nugget
  # and the range line all have something to show.
  set.seed(3)
  n <- 250
  xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000))
  D <- as.matrix(dist(xy))
  xyz <- as.numeric(t(chol(exp(-D / 150) + diag(0.5, n))) %*% rnorm(n))
  pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
  r <- estimate_sac_range(pts, response_var = "z")
  plot(r)

  # A field whose variogram never reaches a sill: the plot still draws it,
  # and the subtitle says why no range is marked.
  xyz$trend <- sin(xyz$x / 400) + rnorm(n, sd = 0.2)
  r2 <- estimate_sac_range(st_as_sf(xyz, coords = c("x", "y"), crs = 32632),
                          response_var = "trend")
  plot(r2)
}
```

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", "coefficients"),
  response = TRUE,
  term = NULL,
  mask = TRUE,
  ...
)
```

Arguments

- | | |
|------|---|
| x | A <code>spatial_fit</code> . The residuals drawn are <code>residuals(x)</code> ; for a custom subclass with no <code>residuals()</code> method (optional, see new_spatial_fit()) they are the response minus <code>fitted(x)</code> , which is what the built-in backends' methods return. |
| type | <p>One of:</p> <ul style="list-style-type: none"> "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, and (unless <code>response = FALSE</code>) the variogram of the response itself on the same points and lags, drawn hollow with a dashed fit. The gap between the two curves is the spatial structure the model absorbed: a residual sill well below the response sill means most of it, two curves that coincide mean none. When both effective ranges were identified over the same point pairs the caption gives the residual sill as a share of the response sill and the two ranges; when either variogram has no identified range, or the two are not over the same pairs, the caption says so and compares nothing: a sill the data never reached is not a number to divide by, and one direction's sill is not comparable with all directions'. The residual range is expected to come out shorter and the residual sill lower even when the model is right, because residuals of a fitted trend understate the variogram (see estimate_sac_range(), "Detrending and the residual-variogram bias"). 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). Each curve is the variogram <code>estimate_sac_range()</code> returns for its variable: all point pairs, or, when the all-pairs fit was unusable, the widest of four directions, which is named with its azimuth (in the title for the residuals, in the caption for the |

response). A fit that identified no range says why in the subtitle, since the overlaid model line is then not a fit to believe. Requires 'gstat'.

"coefficients" For a GWR fit only: the local coefficient of one term mapped at the training locations, which is the reason to fit GWR at all. Locations where the local design is collinear for that term are drawn hollow and grey (mask = TRUE): for a slope, where the kernel-weighted window's slope condition index (cn_slopes, predictors centred in the window) is above 30 or the window is singular, or where the index with the intercept (cn) is above 1e6; for the Intercept, where cn is above 30 (a predictor far from 0 against its local spread makes the local intercept an extrapolation). They are masked because the smooth surface a naive map draws over them is the picture of an unstable estimate, not of a relationship; the subtitle counts them. The condition indices are the fit's `info$local_collinearity`, computed for every location when the model was fitted. A diverging scale centred on zero is used when the coefficient changes sign, otherwise a sequential one.

response	Logical, default TRUE: for type = "variogram", overlay the response's own variogram. Ignored by the other types.
term	For type = "coefficients": which local coefficient to map, one of the names <code>coef(x)</code> returns. Default NULL: the first predictor. Ignored by the other types.
mask	For type = "coefficients": whether to draw locations whose local design is collinear for that term (see type = "coefficients") as hollow grey points instead of colouring them by a coefficient that is not to be believed there. Default TRUE. Locations whose coefficient is non-finite are masked either way.
...	Ignored.

Value

A ggplot object.

See Also

`coef.gwr_fit()` for the coefficients themselves.

Other plotting: `plot.aoa()`, `plot.block_size_sweep()`, `plot.feature_selection()`, `plot.gwr_model_selection()`, `plot.resolution_profile()`, `plot.sac_range()`, `plot_calibration()`, `plot_cv_metrics()`, `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)
  # price depends on elevation, which the forest sees, and on a spatially
  # correlated field it does not: that field is what the residual plots
  # are there to find.
  set.seed(2)
  n <- 120
```

```

xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
                 elev = rnorm(n))
D <- as.matrix(dist(xy[, c("x", "y")]))
xy$price <- 10 + 2 * xy$elev +
  as.numeric(t(chol(exp(-D / 100) + diag(0.3, n))) %*% rnorm(n))
pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
fit <- fit_rf_model(pts, "price", "elev", num_trees = 100, seed = 1)
# print() each one: inside a braced block only the last value is drawn.
print(plot(fit, type = "residuals"))      # structure left in the residuals
print(plot(fit, type = "observed_predicted"))
if (requireNamespace("gstat", quietly = TRUE))
  plot(fit, type = "variogram")          # residual and response variograms compared
}

```

plot_calibration

Plot the interval calibration of a Bayesian cross-validation

Description

`cv_bayes()` scores every fold's posterior predictive intervals at each of `coverage_levels`: the share of held-out observations the interval of that nominal level contained. One coverage number cannot show a pattern; the pairs can. This draws observed coverage against nominal with the diagonal, one point per level for the fold-weighted pooled value (from `predictive_coverage`) and one faint point per fold, so systematic over-confidence (points below the line) or intervals wider than they need to be (above it) are read at a glance. Three levels is a thin curve; pass `coverage_levels = seq(0.1, 0.9, by = 0.1)` to `cv_bayes()` for a full one. Each level is drawn at the nominal value `cv_bayes()` records in `coverage_levels`, so whatever was computed is drawn where it belongs (0.975 at 0.975, not rounded).

Usage

```
plot_calibration(cv, ...)
```

Arguments

<code>cv</code>	The list returned by <code>cv_bayes()</code> , or a <code>compare_models_cv()</code> result that ran the Bayesian backend (its <code>\$bayes_cv</code> is used).
<code>...</code>	Ignored.

Value

A ggplot object.

See Also

Other plotting: `plot.aoa()`, `plot.block_size_sweep()`, `plot.feature_selection()`, `plot.gwr_model_selection()`, `plot.resolution_profile()`, `plot.sac_range()`, `plot.spatial_fit()`, `plot_cv_metrics()`, `plot_folds()`

Examples

```
## Not run:
# Needs brms and a Stan toolchain, and takes minutes: run it, do not check it.
library(sf)
set.seed(1)
n <- 80
dat <- st_as_sf(
  data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
    a = rnorm(n)),
  coords = c("x", "y"), crs = 32632)
dat$z <- 2 * dat$a + rnorm(n)
# Nine coverage levels give a curve rather than three points; chains and
# iterations are kept small to keep this to a few minutes, so the intervals
# will be rough.
cv <- cv_bayes(dat, "z", "a", k = 3, coverage_levels = seq(0.1, 0.9, 0.1),
  fit_args = list(chains = 2, iter = 1000))
plot_calibration(cv)

## End(Not run)
```

plot_cv_metrics

Plot one cross-validation metric fold by fold

Description

A pooled RMSE of 3.2 can come from 3.2 in every fold or from 1.1 in eight folds and 14 in one (a model that works, and a model that fails in one region), and the pooled number cannot tell the two apart. This draws the metric of each fold as a point, sized by the number of held-out predictions the fold contributed, with the pooled value from overall as a horizontal line, so the spread behind the number is visible. A `compare_models_cv()` result draws one panel per model on a shared scale, folds aligned, which is the comparison the shared fold set was built for.

Usage

```
plot_cv_metrics(cv, metric = "RMSE", ...)
```

Arguments

cv	The list returned by <code>cv_spatial()</code> , <code>cv_gwr()</code> , <code>cv_bayes()</code> , <code>cv_rf()</code> or <code>compare_models_cv()</code> .
metric	Character(1) naming a column of <code>fold_metrics</code> (or of <code>by_fold</code>). Default "RMSE".
...	Ignored.

Details

Any column of `fold_metrics` can be drawn, including a backend's extras (bandwidth, CRPS, coverage_95) and columns a metrics function added. A column that is NA in every fold is refused with a message saying why rather than drawn as an empty panel: Adj_R2 is NA for every backend unless p was passed to `cv_spatial()`, by design. The pooled line is drawn from overall

when it carries the metric, from predictive_coverage for cv_bayes()'s coverage and CRPS columns, and not at all for a per-fold extra that has no pooled counterpart (a bandwidth) or for a count (n_pred, n_MAPE, n_SMAPE), whose overall value is the total over the folds; the caption says which. A model with no finite per-fold value gets no panel and no pooled line, and the caption names it.

Value

A ggplot object.

See Also

Other plotting: [plot.aoa\(\)](#), [plot.block_size_sweep\(\)](#), [plot.feature_selection\(\)](#), [plot.gwr_model_selection\(\)](#), [plot.resolution_profile\(\)](#), [plot.sac_range\(\)](#), [plot.spatial_fit\(\)](#), [plot_calibration\(\)](#), [plot_folds\(\)](#)

Examples

```
if (requireNamespace("ranger", quietly = TRUE) &&
    requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 150
  dat <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              a = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  dat$z <- 2 * dat$a + 0.003 * (st_coordinates(dat)[, 1] - 5e5) +
    rnorm(n, 0, 0.5)
  cv <- cv_rf(dat, "z", "a", k = 5, num_trees = 100)
  plot_cv_metrics(cv, "RMSE")
}
```

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. For "block_kfold" folds the block outlines are drawn too, from folds\$params\$blocks, so a fold can be seen to be one region or several and an empty block can be seen to be empty. The subtitle states the parameter that decides whether the scheme leaks, read from folds\$params: the block size (and the estimated range when auto_range found one), the buffer, the number of location groups, or the median NNDM exclusion.

Usage

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

Arguments

<code>folds</code>	A list returned by <code>make_folds()</code> .
<code>points_sf</code>	The <code>sf</code> layer the folds were built from.
<code>boundary</code>	Optional polygonal <code>sf</code> / <code>sfc</code> to draw underneath.
<code>blocks</code>	Logical; draw the block polygons when <code>folds</code> carries them. Default <code>TRUE</code> .

Value

A `ggplot` object.

See Also

Other plotting: [plot.aoa\(\)](#), [plot.block_size_sweep\(\)](#), [plot.feature_selection\(\)](#), [plot.gwr_model_selection\(\)](#), [plot.resolution_profile\(\)](#), [plot.sac_range\(\)](#), [plot.spatial_fit\(\)](#), [plot_calibration\(\)](#), [plot_cv_metrics\(\)](#)

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 80
  pts <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + 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 = NULL,
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	Name of the COLUMN in tessellation_sf to map to fill; NULL for no fill. A numeric column gets a continuous scale, as does a Date or POSIXct column (on a date or time axis) and a units or difftime column (drawn as numbers, with the unit in the legend title); anything else a discrete one. fill_col and label_col name columns, while outline_col, features_col, seeds_col and boundary_col are colours.
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	Colour and line width of the tessellation outline.
features_col, features_size	Colour and point size of the feature overlay.
seeds_col, seeds_size	Colour and point size of the seed overlay.

boundary_col, boundary_size	Colour and line width of the boundary outline.
labels	Logical; draw per-cell labels. Default FALSE.
label_col	Name of the COLUMN holding the label text. Default NULL: the first of "grid_id", "cell_id", "poly_id", "polygon_id" and "id" the layer has, so the cells of build_tessellation() (cell_id) and the output of summarize_by_cell() (poly_id) are labelled without naming one. A units or difftime column is drawn formatted, with its unit.
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 <code>ggplot2::theme_void()</code> . The default is resolved inside the function rather than in the formals, so that a <code>Suggests</code> package is never evaluated before the <code>requireNamespace()</code> 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 instead of being 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_grid_polygons_cached\(\)](#), [create_voronoi_polygons\(\)](#), [ensure_stable_poly_id\(\)](#), [get_voronoi_seeds\(\)](#), [voronoi_seeds_kmeans\(\)](#), [voronoi_seeds_random\(\)](#)

Examples

```
if (requireNamespace("ggplot2", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 60
  pts <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              z = rnorm(n)),
```

```

    coords = c("x", "y"), crs = 32632
  )
  # Cells from the points, the response aggregated onto them, and the map
  # shaded by the cell mean rather than by an arbitrary ID.
  tess <- build_tessellation(pts, method = "voronoi", quiet = TRUE)
  cells <- summarize_by_cell(assign_features_to_polygons(pts, tess$cells),
                             response_var = "z", cells_sf = tess$cells)
  plot_tessellation_map(cells, features_sf = pts, fill_col = "resp_mean_z",
                        legend_title = "mean z")
}

```

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

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

Value

Numeric vector of length `nrow(newdata)`, or a `n_draws x nrow(newdata)` matrix when `draws = TRUE`. If the posterior draw fails the result is all NA (a 1-row matrix for `draws = TRUE`) and the cause is logged. An ordinal or categorical family is not a failed draw and is an error under type = "epred": its expected value is a probability per response category, not one number per row. Use type = "predict", draws = TRUE for posterior draws of the predicted category, as category indices; the share of draws in each category estimates its probability. Without draws = TRUE, type = "predict" returns the mean (or median) category index, an expected rank for an ordinal family and an error for `brms::categorical()`, whose categories have no order. 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 held at its fitted value

`brms` 2.17 to 2.22 do 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, and when `newdata` reaches past the training range the `c` of the `gp()` term is scaled down by as much as the range grew, so `brms` rebuilds exactly the boundary the model was fitted with. A prediction therefore does not depend on which other rows share the call: chunked, fold-wise and single-call predictions agree, and `predict_surface()` does not depend on `chunk_size`. `brms` 2.23.0 and later store L and reuse it, so there `c` is left alone and the two extra rows change nothing.

Predictions outside the training coordinate envelope are extrapolation (a notice is written to the log). A row further than L from the centre of the training coordinates on either axis is past the edge of the basis, where the approximate GP is an odd reflection of the fitted surface rather than an estimate of anything, so it is returned as NA (a column of NA with `draws = TRUE`) with a warning. The default boundary factor puts that edge well outside the training data, so only `newdata` reaching far past it is affected.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

predict.gwr_fit

Predict from a GWR spatial model

Description

When `newdata` is `NULL`, returns the in-sample fitted values. Otherwise estimates the local coefficients at each new location with `GWmodel::gwr.basic(regression.points = )`, with the fit's

kernel and bandwidth (an adaptive bandwidth counts neighbours among the training points), and returns $x^\top \hat{\beta}(u)$. These are the values `GWmodel::gwr.predict()` returns, without its prediction variance, which this method never returned and which costs time cubic in the number of training points. Each location stands alone: one that cannot be estimated does not affect the others. `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

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

Value

Numeric vector aligned to `nrow(newdata)`, with NA for rows dropped as missing or non-finite, and for locations whose local regression cannot be estimated (too few training points within a fixed bandwidth, or a singular local design); a warning counts those. If the design matrix for `newdata` cannot be built, 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.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

<code>predict.rf_fit</code>	<i>Predict from a random forest fit</i>
-----------------------------	---

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 <code>rf_fit</code> .
newdata	Optional <code>sf</code> 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 <code>levels()</code> : 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: <code>ranger</code> 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. (<code>ranger</code> sees a logical as 0/1, so either form is accepted for one.)
...	Passed to <code>ranger</code> 's <code>predict</code> method. Arguments that make <code>ranger</code> return a matrix (<code>predict.all = TRUE</code> , <code>type = "quantiles"</code> , <code>type = "se"</code> with <code>predict.all</code>) are rejected, because this method's contract is one number per row of <code>newdata</code> . Call <code>predict(fit\$engine, data = ...)</code> directly for those. So is anything <code>ranger</code> 's <code>predict</code> method itself refuses, such as <code>type = "quantiles"</code> on a forest grown without <code>quantreg = TRUE</code> or <code>type = "se"</code> without <code>keep.inbag = TRUE</code> : the error names <code>ranger</code> 's reason. <code>seed</code> defaults to a constant: an unset seed makes <code>ranger</code> draw one uniform from the global RNG stream per call, so the number of <code>predict()</code> calls a script happens to make (via predict_surface 's <code>chunk_size</code> , 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 (so all NA, with a WARN line in the log, when every row is). A failure inside `ranger`'s `predict` method is an error, not an all-NA vector.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

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 <code>spatial_fit</code> (e.g. from <code>fit_gwr_model()</code> or <code>fit_bayesian_spatial_model()</code>).
grid	Optional <code>sf</code> POINT layer to predict onto. When <code>NULL</code> , 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. A grid still without a CRS after that is treated as boundary is: taken as EPSG:4326 and reprojected when its coordinates look like lon/lat, otherwise stamped with the fit's CRS, with a warning either way. A grid of polygons (<code>create_grid_polygons()</code> output, say) is reduced to one representative point per cell, as <code>coerce_to_points()</code> does, so covariates are taken at the location predicted for; boundary then keeps the cells whose point falls inside it.
cell_size	Grid resolution in CRS units. Ignored when grid is supplied; when <code>NULL</code> , derived from <code>n_cells</code> . 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. The grid is centred on the training bounding box and covers it: when the extent is not a whole number of cells, it overhangs the box by less than one cell, split evenly between the two sides. A <code>cell_size</code> wider than the extent yields a single centred cell.
n_cells	Approximate cell count used to derive <code>cell_size</code> . 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 <code>sf/sfc</code> ; grid points outside it are dropped. Put through the same CRS replay and reprojection as grid. One still without a CRS after the replay is taken as EPSG:4326 and reprojected when its coordinates look like lon/lat, and is otherwise stamped with the fit's CRS, with a warning either way.
covariates	Optional <code>sf</code> 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. Aligned to the fit's CRS as grid is, with the same warning when it has no CRS.

chunk_size	Rows per prediction call. Default 5000. A pure performance knob: rows do not interact, and for a bayesian_fit the GP boundary is held at its fitted value whatever the chunk holds; see predict.bayesian_fit .
se	Logical; also return a standard-error/posterior-SD column where the backend supports it. Default FALSE. For a bayesian_fit this is the SD of the posterior draws predict() returns, and those are of the expected value by default (type = "epred"): the uncertainty of the mean surface, not of a new observation, which also carries the observation noise. For the predictive SD, the one that goes with prediction intervals and cv_bayes()'s calibration, pass type = "predict" as well.
...	Passed to predict(), e.g. type = "predict" for a bayesian_fit. Not draws, which this function sets itself and refuses here.

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; one a supplied grid already carried, from an earlier surface, is removed otherwise). 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. That is 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.

See Also

Other prediction: [area_of_applicability\(\)](#)

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 = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000),
              elev = rnorm(n)),
    coords = c("x", "y"), crs = 32632
  )
  pts$price <- 10 + 0.01 * (st_coordinates(pts)[, 1] - 5e5) +
```

```

    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)
print(surf[".pred"])      # one prediction per grid cell, as an sf layer
# Check where that surface is extrapolating before mapping it. The grid
# took its covariates from the nearest observation, so here nothing is
# outside; a grid with its own covariate raster is where this bites.
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. Any Z or M coordinate (POINT Z from a GPS, a GeoPackage or KML) is dropped, because every backend works in 2-D map distance.

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

<code>data_sf</code>	An sf object.
<code>response_var</code>	Response variable column name.
<code>predictor_vars</code>	Predictor column names. May be <code>character(0)</code> for an intercept-only model (<code>fit_bayesian_spatial_model()</code> supports one; the GWR and random-forest backends do not and reject it themselves).
<code>boundary</code>	Optional sf/sfc for CRS alignment.
<code>pointize</code>	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".
<code>require_response</code>	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. 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. A name R parses as an expression would fit a different model from the one requested while the fit object still recorded the name you asked for: "B5-B4" is B5 - B4, and "log(a)" is a function call. Rename the column (for example with `make.names()`) before fitting.

Value

An sf object with 2-D (XY) POINT geometry, cleaned of rows carrying missing or non-finite values in the modelling columns or in the coordinates. What was removed is recorded on the attribute "dropped", a list with `n` (rows dropped), `n_geometry` (how many of them for an empty or non-finite geometry), `which` (their positions in `data_sf`), `row_id` (their `.row_id` values when the layer carries that column, else NULL) and `reason` (one per dropped row: "geometry", "missing" or "non_finite", in that order of precedence when several apply), plus `n_rows`, the number of rows returned, which the record was made for. Every fit stores `n` as `$info$n_dropped`. The record describes the rows this call returned and does not survive subsetting: `clean[i, ]`, like `dplyr::filter()`, `slice()` or `arrange()` of it, is a plain layer with no "dropped" attribute, and a fit given such a subset with `.already_prepped = TRUE` reports `n_dropped = 0` even when the parent layer dropped rows. `sf::st_drop_geometry()` keeps the record, since the rows are the same; see [\[.spatialkit_rows\]](#) for what binding such data frames does. 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 either 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\(\)](#), [gp_lengthscales_bounds\(\)](#), [new_spatial_fit\(\)](#)

Other spatial data preparation: [clip_target_for\(\)](#), [coerce_to_points\(\)](#), [ensure_projected\(\)](#), [harmonize_crs\(\)](#)

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
)
clean <- prep_model_data(dat, "resp", "pred") # drops rows 3 (NA) and 5 (Inf)
attr(clean, "dropped")
```

print.aoa	<i>Print an area-of-applicability result</i>
-----------	--

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.

See Also

Other print methods: [print.gwr_model_selection\(\)](#), [print.morans_i\(\)](#), [print.resolution_profile\(\)](#), [print.sac_range\(\)](#)

print.gwr_model_selection	<i>Print a GWR model selection result</i>
---------------------------	---

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. Check the gap 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.

See Also

Other print methods: [print.aoa\(\)](#), [print.morans_i\(\)](#), [print.resolution_profile\(\)](#), [print.sac_range\(\)](#)

print.morans_i	<i>Print a residual Moran's I result</i>
----------------	--

Description

Shows the statistic, the null its moments come from and the evidence behind them. The weight matrix the result carries gets a one-line description in place of the matrix: it is $n \times n$, and autoprinting it buried the statistic under a thousand lines of matrix.

Usage

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

Arguments

x	An object of class morans_i, from residual_morans_i() .
...	Ignored.

Value

x, invisibly.

See Also

Other print methods: [print.aoa\(\)](#), [print.gwr_model_selection\(\)](#), [print.resolution_profile\(\)](#), [print.sac_range\(\)](#)

```
print.resolution_profile
```

Print a resolution profile

Description

Print a resolution profile

Usage

```
## S3 method for class 'resolution_profile'
print(x, digits = 3L, ...)
```

Arguments

x	A resolution_profile.
digits	Significant digits for the table. Default 3.
...	Ignored.

Value

x, invisibly.

See Also

Other print methods: [print.aoa\(\)](#), [print.gwr_model_selection\(\)](#), [print.morans_i\(\)](#), [print.sac_range\(\)](#)

```
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.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`, `summary.spatial_fit()`

<code>print.sac_range</code>	<i>Print a spatial autocorrelation range</i>
------------------------------	--

Description

Prints the effective range as a plain number, with the directional fit summarised beneath it when one is available. A direction whose fit was unusable is labelled with why (`directional_status`) and the range its fit reported (`directional_fitted`) when the object carries them, and unidentified otherwise. A last line names the unit and the CRS the range is a length in (`attr(x, "crs")`), which for lon/lat input is the projected CRS the estimate chose; for a layer with no CRS, it says the range is in that layer's own coordinate units), and whether the variogram is of the response or of its residuals on `predictor_vars` (`detrended`, `detrend_method`).

Usage

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

Arguments

x	An object of class <code>sac_range</code> .
...	Ignored.

Value

x, invisibly.

See Also

Other print methods: `print.aoa()`, `print.gwr_model_selection()`, `print.morans_i()`, `print.resolution_profile()`

print.spatial_fit	<i>Print a fitted spatial model</i>
-------------------	-------------------------------------

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

<code>x</code>	A <code>spatial_fit</code> object.
<code>...</code>	Ignored.

Value

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

See Also

Other methods on a fitted model: [coef.bayesian_fit\(\)](#), [coef.gwr_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.gwr_fit\(\)](#), [fitted.rf_fit\(\)](#), [model_metrics\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [residuals.bayesian_fit\(\)](#), [residuals.gwr_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

residual_morans_i	<i>Compute Moran's I on the residuals of a fitted spatial model</i>
-------------------	---

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"),
  keep_weights = FALSE
)
```

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 $n \times n$ 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, with no tie-break 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 <code>spdep's knearneigh() + nb2listw(style = "W")</code> 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 <code>weights</code> argument that is not an $n \times n$ matrix is an error.
k	Integer number of nearest neighbours used when building the default weight matrix (ignored when <code>weights</code> is supplied). Default 8. $k \geq n - 1$ 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 <i>and</i> 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 logs a warning (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 <code>df</code> in the result is then $n - 1$. Read <code>null</code> in the result to see which it was. See Which null, and when it is approximate above.

keep_weights Logical. Return the $n \times n$ weight matrix in the result? Defaults to FALSE: the matrix dominates the object's size (50.3 KB of a 53.5 KB result at $n = 500$ in its sparse form, and 191 MB at the $n = 5000$ the dense fallback is capped at) while most uses read only the statistic and its moments, and `weights_summary` says what it was. Set TRUE when you need the matrix itself.

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", $(n/S_0)\text{tr}(MW)/(n-p)$ 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 it, 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".

weights The $n \times n$ weight matrix the statistic was computed with, either the row-standardised k-nearest-neighbour matrix built here (sparse when **Matrix** is installed) or the supplied weights after the diagonal was zeroed. It is NULL unless `keep_weights = TRUE`.

kurtosis The residual kurtosis m_4/m_2^2 , which the randomisation variance conditions on and which says how far the residuals sit from the Gaussian case the residual moments assume (3 for a normal sample).

p The rank of the rebuilt design matrix behind the "residual" moments; NA under "randomisation".

exact Logical: TRUE when the moments are exact for these residuals (the residual null on OLS residuals of the response on the rebuilt design), FALSE when they are an approximation (the residual null forced onto a non-OLS backend, or the randomisation null, whose exchangeable moments model residuals do not satisfy).

weights_summary What the weight matrix was, present whether or not the matrix itself was kept: n, storage (its class), neighbours (the smallest and largest number of neighbours any row has, and NA for a dense matrix, where counting them would allocate a second one), kept (whether `weights` holds the matrix) and desc, the one line `print()` shows.

The list is classed "morans_i" and has a `print()` method, so the console shows the statistic and its null without printing the $n \times n$ weights matrix; `[]` drops the class, and `$`, `[[` and `unlist()` are unaffected. A custom fit whose class has no `residuals()` method (which `new_spatial_fit` calls optional) is scored on the observed response minus `fitted()`, as `plot()` does for it. Returns

NULL with a warning if computation fails, saying why: `residuals()` raised an error (its message is quoted), or returned NULL and the response minus `fitted()` could not be formed either (the reason is quoted), fewer than 4 valid residuals, or a residual vector whose length does not match the fit's `data_sf`.

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% of the time where the correct one rejected 31%.

"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`:

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

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

These assume normal errors and do not condition 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% against a nominal 5%. They agree with `spdep::lm.morantest()` to machine 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 instead of 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 without 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\(\)](#), [model_metrics\(\)](#)

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 = 5e5 + runif(n, 0, 1000), y = 5e6 + 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$price <- 10 + 0.02 * (st_coordinates(dat)[, 1] - 5e5) +
    2 * dat$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)
}
```

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.

See Also

Other methods on a fitted model: [coef.bayesian_fit\(\)](#), [coef.gwr_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.gwr_fit\(\)](#), [fitted.rf_fit\(\)](#), [model_metrics\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [print.spatial_fit\(\)](#), [residuals.gwr_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

residuals.gwr_fit	<i>In-sample residuals from a GWR fit</i>
-------------------	---

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.

See Also

Other methods on a fitted model: [coef.bayesian_fit\(\)](#), [coef.gwr_fit\(\)](#), [coef.rf_fit\(\)](#), [fitted.bayesian_fit\(\)](#), [fitted.gwr_fit\(\)](#), [fitted.rf_fit\(\)](#), [model_metrics\(\)](#), [predict.bayesian_fit\(\)](#), [predict.gwr_fit\(\)](#), [predict.rf_fit\(\)](#), [print.rf_fit\(\)](#), [print.spatial_fit\(\)](#), [residuals.bayesian_fit\(\)](#), [residuals.rf_fit\(\)](#), [summary.spatial_fit\(\)](#)

residuals.rf_fit	<i>Out-of-bag residuals from a random forest fit</i>
------------------	--

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 <code>rf_fit</code> .
...	Ignored.

Value

Numeric vector of length `object$n`.

See Also

Other methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `summary.spatial_fit()`

resolution_profile	<i>Score every candidate number of cells on several criteria at once</i>
--------------------	--

Description

A tessellation's resolution is the number of cells the points are cut into, and no single number settles it: the within-cluster sum of squares of the coordinates has an elbow, a piecewise-constant approximation of the response has a Mallows C_p , the residual autocorrelation of the cell means has a z , and the cell means have a reliability. This function computes all of them at every level of a ladder the data bound, and returns the table, so that the level chosen, by `select_resolution()` or by eye, can be defended with the whole profile in place of one criterion's argmin. Mallows (1973) presents C_p itself as a display of the bias-variance trade-off across candidates; he does not offer it as a rule that picks one. This is that display, with the other criteria alongside.

Usage

```
resolution_profile(
  data_sf,
  response_var = NULL,
  predictor_vars = NULL,
  levels = NULL,
  n_levels = 20L,
  min_cell_n = 9L,
  sample_n = 1500L,
  nstart = 25L,
  seed = 123L,
  sac = NULL,
  select_on = c("all", "split"),
  range_floor = TRUE
)
```

Arguments

<code>data_sf</code>	An sf object of points (other geometries are reduced to representative points). Features with empty or non-finite coordinates are dropped with a warning.
<code>response_var</code>	Optional response column name (numeric or logical). Enables <code>cp</code> and <code>moran_z</code> . A variogram estimated from it also sets the floor of the ladder and <code>reliability</code> . Rows where it, or a predictor, is missing or non-finite stay in the geometry and are left out of the OLS fit, the RSS, <code>cp</code> and <code>moran_z</code> ; a logged warning gives their number.
<code>predictor_vars</code>	Optional predictor column names (numeric or logical). With them, <code>cp</code> scores the OLS residuals of the response on the predictors, the variogram is estimated from those residuals, and <code>moran_z</code> regresses the cell means on the cell-mean predictors.
<code>levels</code>	Optional integer vector of level counts to score, replacing the ladder; values below 2, above the number of distinct locations, or at or above the number of points, are dropped (k-means cannot place more centres than there are distinct points, and <code>stats::kmeans()</code> refuses as many centres as points).
<code>n_levels</code>	Number of levels on the ladder. Default 20.
<code>min_cell_n</code>	Minimum average number of points per cell that a level must keep; sets the ceiling. Default 9.
<code>sample_n</code>	Points are subsampled to this many before the k-means fits, as in <code>determine_optimal_levels()</code> . Default 1500. The columns read off the fitted cells (<code>wss</code> , the <code>cell_</code> columns, <code>rss</code> , <code>moran_i</code> , <code>moran_z</code>) describe the subsample; the bounds, supported, the variance term of <code>cp</code> and <code>reliability</code> describe every point of the layer, so the answer does not change with <code>sample_n</code> except through the fits.
<code>nstart</code>	k-means++ restarts per level. Default 25.
<code>seed</code>	RNG seed for the subsample and the restarts; restored afterwards. Default 123. The rows are put in coordinate order before either, so the profile does not depend on the order they come in.

sac	Optional <code>sac_range</code> object from <code>estimate_sac_range()</code> to take the range, nugget and correlation function from. Pass one fitted with <code>detrend = "reml"</code> , say. It must describe the variable the profile scores: the raw response without <code>predictor_vars</code> , the residuals on them with; a sac whose detrended attribute says otherwise is used with a warning. Its range is read in its own CRS (<code>attr(sac, "crs")</code>), to which the points are transformed first, so the area, the floor and <code>cell_diam_median</code> are then in that CRS's units. A sac whose range was rejected (NA with a <code>rejected_reason</code>) gives <code>cp</code> its nugget, with a warning, and leaves <code>reliability</code> NA, since that needs the range; one whose model did not converge, or whose range is below the shortest lag fitted (a structure that cannot be told from a nugget, so the nugget is not identified either), gives neither, and <code>cp</code> takes its noise variance from the finest level (see <code>cp</code> above). Whether the model converged is read from the model (<code>attr(attr(sac, "variogram_model"), "converged")</code>), which <code>estimate_sac_range()</code> sets), not from the reason: the reason names one ground for the refusal, and a fit that did not converge is most often refused as past the largest lag. The warning then names both. A sac that carries no such flag (one made by hand, say) is judged by its reason. Under <code>select_on = "split"</code> the sac must come from the selection half alone: run the profile once without it, fit the sac on <code>data_sf[attr(p, "split")\$selection,]</code> and pass it to a second call with the same seed, which makes the same split. When NULL and a response is given, one is estimated on the subsample (its selection half under "split") with the same <code>predictor_vars</code> . A plain number is taken as the range alone, in the units of the CRS the profile is computed in (metres for lon/lat input): it sets the floor, <code>reliability</code> , which needs a fitted model, is NA, and <code>cp</code> takes its noise variance from the finest level. A units object is refused rather than read as a number in whatever unit it was written in.
select_on	"all" (default) profiles every point; "split" reads the response on one spatially blocked half only (the OLS fit, the variogram, <code>rss</code> , <code>cp</code> and <code>moran_z</code>) and returns the other half as the set to estimate on, in the "split" attribute. The cells, <code>wss</code> , <code>elbow</code> , and the extent and point counts behind the bounds, <code>cp</code> 's variance term and <code>reliability</code> still come from every point, because the tessellation the count is for is built on every point: the levels are cell counts for the whole layer, and the estimation half's response never touches them. See the "Post-selection inference" section of determine_optimal_levels ; the profile reads the response whenever <code>response_var</code> is given.
range_floor	TRUE (default) starts the ladder at the range floor when the data support it; FALSE starts it at 2 whatever the range, and the floor is only reported in the bounds and the print. Use FALSE to compare profiles whose range estimates differ (see "The ladder and its bounds").

Value

A data.frame of class `resolution_profile` with one row per level and columns `levels`, `wss`, `wss_spread` (relative spread of WSS across the restarts), `elbow`, `cell_n_min`, `cell_n_median`, `cell_diam_median` (twice the median RMS radius of the cells, in coordinate units: about 0.8 of the side of a square cell of the same area, and less for finer cells, so a size to compare levels by rather than a width), `rss`, `cp`, `moran_i`, `moran_z` and `reliability`; columns a missing input leaves undefined are NA. Attributes: `bounds` (a list with `floor`, `ceiling`, `ceiling_from`

("min_cell_n", "distinct locations" or "sample_n", whichever bound it), supported, area, range, n (the points in the layer), n_sample (the points the k-means fits ran on), n_distinct, min_cell_n, range_floor), variogram (a list with nugget, psill, range (NA when rejected), model and detrended (whether the sac says it is a variogram of residuals; NA when it does not say); NULL when none was usable), variable ("response", "residuals" or NA), wss_bumps, nstart, sac (the range object used) and, with select_on = "split", split (a spatialkit_split: selection and estimation, integer row positions in data_sf, with the method and seed that made them), cp_noise (a list with source, "variogram nugget" or "finest-level residual mean square", value, the τ^2 used, and level, the level it was read at or NA; NULL when cp has neither), centres (a list named by level of the two-column matrices of the scored partitions' centres) and centres_crs (the CRS they are in). A k-means partition is the Voronoi partition of its centres, so these seed the very cells each level was scored on: select_resolution() returns the chosen level's as \$seeds, and get_voronoi_seeds() and voronoi_seeds_kmeans() use them when given the selection or the profile.

The ladder and its bounds

Cells are k-means clusters of the (projected) coordinates, fitted at each level as the best of nstart k-means++ restarts (see [determine_optimal_levels](#) for why). Levels are spaced logarithmically, because cell diameter scales as $L^{-1/2}$: a unit step wastes fits at large L and starves resolution at small. The ladder runs from a floor to a ceiling the data impose. The ceiling is floor($n / \text{min_cell_n}$), with n every point of the layer (not the subsample): cells with fewer than min_cell_n points on average have too little support, and Moran's z is not computable at nine cells or fewer in any case. It is also held to the number of distinct locations, one cell on each, and to one short of the number of points, which stats::kmeans() needs. The floor is ceiling($\text{area} / \text{range}^2$) when an autocorrelation range is available: cells wider than the range average over more than one patch of the field. When the floor exceeds the ceiling the data cannot support a tessellation that respects their own correlation structure; that is reported as a finding (a logged warning, and attr(x, "bounds")\$supported is FALSE) and the ladder runs from 2 to the ceiling anyway, so the profile still shows what each level costs. On a layer larger than sample_n the ceiling is also held to half the subsample (two subsample points per cell, the least a fitted cell can be scored on); ceiling_from is then "sample_n", a floor above that is logged with a request to raise sample_n, and it does not make supported FALSE.

The floor moves with the range estimate, and the ceiling with n , so two profiles of similar data (the folds of a cross-validation, say) can sit on either side of the point where the floor applies: one ladder then starts at the floor and the other at 2, and the criteria that sit near the bottom of the ladder (reliability, which routinely peaks at the floor, and elbow) can differ between them by a factor of 10 or more. To compare profiles, pass the same levels to each, or range_floor = FALSE to start every ladder at 2 while the floor is still reported.

The criteria, and how each behaved when measured

elbow How far the WSS curve sags below a power law: on log-log axes, log WSS below the straight line from $k = 1$ (the total sum of squares) to the last level, in natural-log units (larger is better). Points with no cluster structure have a WSS close to c/k , which is straight on those axes, so the column is NA at every level unless the largest sag reaches log 1.25, and the print says there is no elbow (the rule and its calibration are in [determine_optimal_levels](#)). The classical chord on linear axes found a "knee" on such a layer anyway, at about $\sqrt{L_{\text{first}} L_{\text{last}}}$, where the ladder's ends put it. A level whose WSS is 0 (to within 10^{-12} of the total), one cell

on every distinct location, which the ladder reaches when locations repeat, is left out of the line; when the other levels have no elbow, that fall to zero is the elbow, and its sag is measured with the WSS floored at 10^{-12} of the total, which puts it far above any other level's. Geometry only; it knows nothing of the response.

cp Mallows' C_p of the piecewise-constant approximation of the response (or of its OLS residuals on predictor_vars) by cell means: $RSS(L)/n + 2\tau^2 L/n$, with τ^2 the nugget of the fitted variogram (lower is better). It estimates the error of predicting a new observation by the mean of its cell. When the cells are fitted to a subsample of m of the N points with a response, the penalty is split between the two: $RSS(L)/m + \tau^2 L_m/m + \tau^2 L/N$, where the first two terms estimate the approximation error from the subsample (adding back the optimism of its own cell means, over the L_m cells its scored points fall in) and the last is the variance of cell means built from all N , which is what the tessellation will carry. With no subsample it is the formula above. **Measured on simulated exponential fields (600 points on a 1000-unit extent, sill 1, 20 replicates): with a nugget of 0.3 its minimum sat at the support ceiling in every replicate at effective ranges of 90, 300 and 900; with a nugget of 2 it was interior (median 44, range 8–66).** On a smooth field with little noise the approximation keeps improving as cells shrink and the penalty is too small to stop it, so C_p says "as fine as the support allows" and min_cell_n is what is choosing; select_resolution() says so when that happens. It becomes a genuine interior criterion only when the nugget is a large share of the sill. With a nugget of 0 (under 10^{-4} of the sill; usually a fit clipped at its lower bound) the penalty is 0 and C_p descends to the ceiling whatever the field; the profile warns. When the variogram gives no nugget (none was fitted, its fit did not converge, its range is below the shortest lag, or sac is a range alone), τ^2 is Mallows' own choice instead: the residual mean square $RSS/(m - L_m)$ of the finest level whose cells hold at least two scored rows on average. It counts the structure within those cells as noise too, so on average it is no smaller than the nugget and errs towards fewer cells; the profile warns, and attr(, "cp_noise") records which was used. When no level's cells hold that many, cp is NA at every level.

moran_z The standardised deviate of Moran's I on the residuals of the cell means regressed on the cell-mean predictors (an intercept alone when there are none): how much spatial structure the tessellation has left unexplained ($|z|$ smaller is better). Calibrated and flat in L on a response with no structure (see [determine_optimal_levels](#)), so it separates levels only where structure remains. NA at nine cells or fewer.

reliability The between-cell signal's share of the spread in the cell means, from the fitted variogram alone via Krige's additivity relation (Cressie 1996), for square cells of the level's average area holding the level's average share of the layer's points with a response (larger is better). This is the shrinkage factor of Fay and Herriot (1979). It has an interior optimum, and a broad one: validated against the empirical reliability of true block means on simulated fields, the analytic and empirical optima agreed to within a level or two where the empirical estimate was stable, and the band within 2 percent of the maximum spanned a factor of 3–6 in L . Read the flat region, not the argmax. The domain term is taken over the convex hull the area is measured on, so rotating the layer does not move it. NA without a usable variogram, and when the variogram's range was rejected (see sac).

cp and reliability answer different questions: how well the cells represent the field, and whether the cell values are distinguishable from noise. They can disagree, and both are shown so the choice between them is made knowingly.

References

- Cressie, N. (1996). Change of support and the modifiable areal unit problem. *Geographical Systems*, 3(2–3), 159–180.
- Fay, R. E. and Herriot, R. A. (1979). Estimates of income for small places: an application of James-Stein procedures to census data. *Journal of the American Statistical Association*, 74(366), 269–277. doi:10.1080/01621459.1979.10482505
- Mallows, C. L. (1973). Some comments on C_p . *Technometrics*, 15(4), 661–675. doi:10.1080/00401706.1973.10489103

See Also

[select_resolution\(\)](#) to read a level and its flat region off the profile; [determine_optimal_levels\(\)](#) for the integer-vector interface; [build_tessellation\(\)](#) and [get_voronoi_seeds\(\)](#), which accept the profile or a [select_resolution\(\)](#) result directly as `approx_n_cells` and `n`.

Other aggregation: [assign_features_to_polygons\(\)](#), [determine_optimal_levels\(\)](#), [kriging_adequacy\(\)](#), [select_resolution\(\)](#), [summarize_by_cell\(\)](#), [summary.resolution_profile\(\)](#)

Examples

```
if (requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  # An exponential field with range parameter 200 (true effective range
  # 600 m) on a 1 km square, with a nugget of 0.6 on a unit sill: enough
  # noise for Mallows' Cp to have an interior optimum rather than descend
  # to the ceiling.
  set.seed(4)
  n <- 400
  xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000))
  D <- as.matrix(dist(xy))
  xy$z <- as.numeric(t(chol(exp(-D / 200) + diag(0.6, n))) %*% rnorm(n))
  pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
  prof <- resolution_profile(pts, response_var = "z", n_levels = 12)
  print(prof)
  # one row per level; print() because only the
  # last value of a braced block is shown
  select_resolution(prof, criterion = "cp")
}
```

sac_nugget

The nugget of an estimated autocorrelation range

Description

The nugget variance of the variogram model behind a [estimate_sac_range\(\)](#) result: the semivariance at zero separation, i.e. measurement error plus variation at scales shorter than the first lag bin of the empirical variogram (gstat's bins are cutoff / 15 wide, about `max_dist` / 30 at the default cutoff), which can be far wider than the spacing of close pairs. It is extrapolated to zero from that bin, not observed, and a fit that runs into its lower bound reports exactly 0. It is carried as the nugget attribute of every classed result, identified or rejected, because it is the number a resolution criterion for a tessellation needs (the short-lag variance that no cell can average away).

Usage

```
sac_nugget(x)
```

Arguments

`x` A `sac_range` object, or anything else.

Value

A single number: the nugget in the units of the response's variance; `NA_real_` when `x` carries no fitted model (a bare NA from a run that could not fit anything, a rejected result whose fits were all singular, or an object that is not a `sac_range`).

See Also

[estimate_sac_range](#), which produces the object.

Other cross-validation: [area_of_applicability\(\)](#), [cv_bayes\(\)](#), [cv_block_size_sweep\(\)](#), [cv_gwr\(\)](#), [cv_rf\(\)](#), [cv_spatial\(\)](#), [estimate_sac_range\(\)](#), [fold_separation\(\)](#), [gwr_model_selection\(\)](#), [make_folds\(\)](#), [select_features_forward\(\)](#)

Examples

```
if (requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  # A field with a real nugget: half a unit of white noise on a unit sill.
  set.seed(3)
  n <- 250
  xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000))
  D <- as.matrix(dist(xy))
  xy$z <- as.numeric(t(chol(exp(-D / 150) + diag(0.5, n))) %*% rnorm(n))
  r <- estimate_sac_range(st_as_sf(xy, coords = c("x", "y"), crs = 32632), "z")
  print(sac_nugget(r)) # the fitted nugget variance, on the sill's scale
  sac_nugget(NA)      # nothing fitted: NA
}
```

```
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,
  auto_range = FALSE,
  select_on = c("all", "split")
)
```

Arguments

<code>train_sf</code>	Training data (sf).
<code>response_var</code>	Character(1).
<code>candidate_vars</code>	Character vector of predictors to choose among.
<code>fit_fn</code>	A function (<code>train_sf</code> , <code>predictor_vars</code>) returning a <code>spatial_fit</code> . The signature takes two arguments because selection has to refit with different predictor sets.
<code>k</code>	Inner fold count. Default 5.
<code>method</code>	Inner fold method. Default "block_kfold".
<code>block_size</code>	Passed to <code>make_folds()</code> ; inherit the outer block size so inner and outer blocks are on the same spatial scale.
<code>metric</code>	Score to optimise: "RMSE", "MAE" (minimised) or "R2" (maximised). Default "RMSE".
<code>tol</code>	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 <code>tol</code> bites from step 1, but only when that null model can be scored. Backends that refuse a zero-length <code>predictor_vars</code> (<code>fit_rf_model</code> and <code>fit_gwr_model</code> both do) have no null score, and there the first variable is accepted unconditionally.
<code>max_vars</code>	Optional cap on how many predictors to select.
<code>max_fits</code>	Abort if the sweep would exceed this many model fits. Default 5000.
<code>seed</code>	RNG seed. It governs both the inner fold construction and the cross-validation itself: it is forwarded to <code>cv_spatial(seed =)</code> , which draws one RNG stream per fold from it, so it also seeds the <i>learner</i> inside every fold. A stochastic <code>fit_fn</code> 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.
auto_range	Logical. If TRUE and method is "block_kfold", the autocorrelation range is estimated from the response (detrended on the candidates) and used as the minimum block size of the inner folds, exactly as in make_folds() ; block_size still applies as a floor. Default FALSE, which keeps the geometric blocks. Inner blocks smaller than the range let a candidate be selected for spatial proximity to the response rather than for predicting it, which is the same failure the random_kfold caution above exists to prevent, so the leakage warning make_folds() raises applies here with more force than usual.
select_on	"all" (default) runs the sweep on every row of train_sf. "split" runs it on one spatially blocked half, then fits the selected set on that half and scores it on the other: score_holdout is then the selected model's metric on rows whose response the sweep never read (the sweep's own score is not; see "The score is not a performance estimate"). It is one estimate from one region: the halves share a border with no buffer, so rows near it are still correlated with the selection half. Both halves come back in \$split. See the "Post-selection inference" section of determine_optimal_levels for the trade: coverage for half the sample.

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. 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. The cost grows fast: 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 of class "feature_selection" (so that [plot.feature_selection\(\)](#) draws the selection path) with selected (the chosen predictors, in the order they were added), score, score_holdout, history, params and split. score is the winning set's cross-validated metric at the final step: the **selection-internal** optimum, optimistically biased because it was chosen as the best of many (see the section above), and NA when nothing was selected. history is a dataframe with step, variable, score and n_pred, 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. Every set is scored on the same rows: those the null model's cross-validation predicted or, when there is no null model, those any step-1 set predicted; params\$n_scored counts them (a warning says so when that is fewer than all). n_pred is

how many rows the set's cross-validation predicted. A set that left some of the scored rows unpredicted, because a fold failed for it, has score NA, with a warning naming it: scored on the rows it did predict it would be compared on fewer, usually easier, rows than its rivals. A factor with a level found in one spatial block only is the usual case, and cannot be selected. score_holdout is NA unless select_on = "split", and then the selected set's metric when fitted on the selection half and predicted on the estimation half (R^2 against the selection half's mean, the out-of-sample convention); NA when nothing was selected or the prediction failed. split is NULL or a list with selection and estimation, integer row positions in train_sf as passed; rows the completeness filter above dropped are in neither. params records metric, method, k, tol, seed, auto_range, select_on, n_candidates, estimated_fits and n_scored.

The score is not a performance estimate

\$score is the cross-validated metric of the winning set at the final step: the best of every candidate set the sweep scored. That is the number the selection optimised, and a number optimised over many candidates is optimistically biased by construction: Cawley and Talbot (2010) show the bias can exceed the genuine differences between the models being compared. Quote it as the selection criterion, not as the performance of the selected model. An honest performance estimate needs data the selection never saw. Two ways to get one: run this function inside the fit_fn of `cv_spatial()`, so the outer folds score a model whose predictors were chosen on the inner ones alone; or pass select_on = "split", which selects on one spatial half of train_sf and returns the selected set's score on the other as score_holdout.

References

Cawley, G. C. and Talbot, N. L. C. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11, 2079–2107. <https://jmlr.org/papers/v11/cawley10a.html>

See Also

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

Examples

```
if (requireNamespace("GWmodel", quietly = TRUE) &&
    requireNamespace("sp", quietly = TRUE)) {
  library(sf)
  set.seed(1)
  n <- 120
  pts <- st_as_sf(
    data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + 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
}

```

select_resolution	<i>Read a level, and the region over which it is not distinguishable, off a profile</i>
-------------------	---

Description

Picks the level a criterion prefers, together with the *flat region*: every level whose criterion value is within `tol` of the optimum. On the criteria this package computes the flat region is routinely wide: the reliability curve is flat to within 2 percent over a factor of 3–6 in the number of cells, and C_p on a smooth field descends to the support ceiling. The region is the answer, and the `argmin` only a point in it. When the optimum sits at an end of the levels the criterion was scored at, the result says so and names the bound, because a bound is then doing the choosing rather than the criterion (see [resolution_profile](#) for what each criterion measures and how it behaved on simulated fields).

Usage

```

select_resolution(
  profile,
  criterion = c("cp", "reliability", "elbow", "morán_z"),
  tol = 0.02
)

```

Arguments

<code>profile</code>	A <code>resolution_profile</code> .
<code>criterion</code>	Which column decides: "cp" (minimised), "reliability" (maximised), "elbow" (maximised) or "morán_z" ($ z $ minimised).
<code>tol</code>	Width of the flat region. For cp and reliability it is relative to the optimum's value (0.02 keeps levels within 2 percent of it); for elbow and morán_z, whose optimum can be zero, it is relative to the criterion's range over the ladder.

Value

A list of class `resolution_selection` with `best` (the level), `flat` (the levels in the flat region, ascending; a set, which can skip a rung), `criterion`, `value` (the optimum), `at_ceiling` and `at_floor` (logical: the optimum is the last or first of the levels this criterion was scored at, which for `morán_z` starts above nine cells), `edge` (which bound that is, in words: the support ceiling, the subsample's ceiling, the range floor, the ladder's own end, or the first or last level the criterion is computable at; NA for an interior optimum), `n_levels`, `values` (the criterion at every level, NA where it could not be computed) and `seeds` (an sf POINT layer, with `seed_id`, of the centres of the partition the profile scored at best, in the CRS it was computed in; NULL for a profile that does not carry them). Their Voronoi cells are the cells that were scored; `get_voronoi_seeds`(method = "kmeans", n = <this>) and `voronoi_seeds_kmeans`(k = <this>) return them.

See Also

Other aggregation: [assign_features_to_polygons\(\)](#), [determine_optimal_levels\(\)](#), [kriging_adequacy\(\)](#), [resolution_profile\(\)](#), [summarize_by_cell\(\)](#), [summary.resolution_profile\(\)](#)

Examples

```
if (requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  # An exponential field with range parameter 200 (true effective range
  # 600 m) on a 1 km square, with a nugget of 0.6 on a unit sill: enough
  # noise for Mallows' Cp to have an interior optimum rather than descend
  # to the ceiling.
  set.seed(4)
  n <- 400
  xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000))
  D <- as.matrix(dist(xy))
  xy$z <- as.numeric(t(chol(exp(-D / 200) + diag(0.6, n))) %*% rnorm(n))
  pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
  prof <- resolution_profile(pts, response_var = "z", n_levels = 12)

  sel <- select_resolution(prof, criterion = "cp")
  print(sel)           # the level, and the flat region around it
  print(sel$flat)      # every level within `tol` of the optimum
  print(sel$edge)      # NA here: the optimum is interior

  # Reliability prefers coarse cells and here runs into the floor the
  # autocorrelation range sets, which the result says in words.
  rel <- select_resolution(prof, criterion = "reliability")
  print(rel)
  c(at_floor = rel$at_floor, edge = rel$edge)
}
```

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

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.

While a document is being knitted (R Markdown, Quarto, a **pkgdown** article) the console echo is also sent as an R message, because **knitr** does not capture what is written to the console's error stream and the cautions would otherwise be missing from the output. They appear as `## WARN [...]` lines, and the chunk option `message = FALSE`, like `suppressMessages()`, keeps them out of the document. A line the package also raises as an R warning is not repeated, since the document shows the warning. Outside **knitr** nothing changes.

Value

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

See Also

Other package options and caches: `clear_fitted_cache()`, `clear_grid_cache()`, `create_grid_polygons_cached()`

Examples

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

<code>summarize_by_cell</code>	<i>Summarize features by polygon/cell ID</i>
--------------------------------	--

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,
conf_level = NULL,
area = FALSE
)

```

Arguments

assigned_points_sf	An sf object with a cell identifier column.
response_var	Optional response column name for per-cell aggregation: a single character string. A name that is not a column, or a column that is not numeric, is skipped with a warning.
predictor_vars	Optional predictor column names for per-cell aggregation. Names that are not columns, and columns that are not numeric, are skipped with a warning.
id_col	Preferred name of the polygon/cell ID column.
agg_funs	Named list of aggregation functions. Default <code>list(mean = \ (x) mean(x, na.rm = TRUE))</code> . Additional common options: <code>median</code> , <code>sum</code> , <code>sd</code> . A single function (<code>agg_funs = median</code>) or a character vector of function names (<code>c("median", "sum")</code>) is also accepted. A single function is named after the expression passed: a name gives that name (<code>median</code> , or <code>f</code> for a variable <code>f</code> holding a function), <code>stats::median</code> gives <code>median</code> , and any other expression gives <code>agg1</code> ; pass a named list to choose the name. Anything else falls back to the default <code>mean</code> with a warning.
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 <code>cells_sf</code> , with one row per cell in <code>cells_sf</code> . Cells that no feature fell in are kept, with NA summaries. Duplicate ID values in <code>cells_sf</code> would multiply those rows, so they are reported with a warning. Its ID column is the first of <code>id_col</code> , <code>"poly_id"</code> , <code>"polygon_id"</code> , <code>"id"</code> , <code>"cell_id"</code> and <code>"grid_id"</code> it carries, the list and order <code>assign_features_to_polygons()</code> reads the polygons' IDs from. A summarised ID that matches no cell is reported with a warning, since the join drops it with its points; a <code>cells_sf</code> with none of those columns, or one that is not an sf object, gives a warning and a plain data frame (an error with <code>area = TRUE</code>). When <code>NULL</code> (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 <code>"deff_applied"</code> attribute is attached, so a result carrying none was computed this way. <code>"variogram"</code> Compute a per-cell design effect from a fitted variogram: for <code>n</code> points in a cell with correlation matrix <code>R</code>, the effective sample size of the

mean is $n^2 / \text{sum}(R)$, so $\text{deff} = \text{sum}(R) / n$. 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- ρ 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) and with `gstat`'s 2-D geometric anisotropy (`vgm(..., anis = c(angle, ratio))`), applied as `gstat` applies it. A model of any other family falls back to $\text{deff} = 1$ with a warning naming it, and so does a request with no usable model (none supplied and none could be estimated, or a rejected fit that could not be replaced by an estimate); see "Value" for how to detect a fallback. A model with no structured component (a pure nugget) implies that distinct observations are uncorrelated, so it is applied as a design effect of 1 in every cell, not treated as a fallback. Points with empty geometry count towards their cells' values but not towards the correlation, with a warning.

"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: $\text{deff}_i = 1 + (n_i - 1) * \rho$. The ICC is the ANOVA (method-of-moments) estimator with Donner's n_0 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 ($N - k \geq 2$); the ICC is taken as 0 (no correction for that variable type) otherwise, and likewise when the estimate itself comes out at or below 0. The `"deff_applied"` attribute is attached when either ICC is positive, so when both are 0 there is none.

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 finite number ≥ 1 (including a value below 1, which would *shrink* the standard errors, and NA or Inf) is refused with a warning and replaced by 1.

`sac`

Optional `sac_range` object from `estimate_sac_range()`, used when $\text{deff} = \text{"variogram"}$. 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 value is NA and it carries a `rejected_reason` attribute) carries no usable correlation function, so it is set aside rather than correcting by a shape that was

	not trusted enough to report a range: the variogram is then estimated as if no sac had been given, with a plain warning saying so, or, where that is not possible, deff falls back to 1 with the fallback warning, which names the rejection. A sac with no variogram_model attribute – a plain number or a units object, say – is a range without a correlation function, and is set aside the same way. A sac fitted to residuals (attr(sac, "detrended") TRUE) is used as given, with a warning when it corrects response columns (see "Design effects and variable types").
deff_max_n	Cells with more than this many points are subsampled before forming the $n \times n$ correlation matrix used by deff = "variogram". Default 500. It must be a single number of at least 2 when deff = "variogram"; anything else is an error.
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.
conf_level	Optional confidence level in (0, 1), such as 0.95. When given, every numeric response and predictor column also gets ..neff_*, ..df_*, ..ci_lo_* and ..ci_hi_* (see "Confidence intervals"). Default NULL: no interval columns, and the frame is exactly what it was before this argument existed.
area	Logical, default FALSE. With TRUE, and cells_sf supplied, the result gains cell_area (each cell's planar area in the squared units of cells_sf's CRS) and n_per_area (the count of rows in the cell over that area, a point density; a rate of anything else is that thing's agg_funs sum over cell_area). The request is refused with an error, not answered with a number, when the cells' CRS distorts areas across them by more than 1 percent, measured as the spread of planar-to-geodesic area ratios over the cells: a density is a comparison between cells and is meaningless where the map scale differs from one cell to the next. Inside a UTM zone the spread is under 0.3 percent and the request goes through; the conterminous United States forced into one zone (14 percent), or a few degrees of latitude in Web Mercator (4 percent at 48N), does not. ensure_projected() with purpose = "area" chooses an equal-area CRS for lon/lat input; build the cells in it. Lon/lat cells are measured geodesically instead: their cell_area is sf::st_area()'s area in square metres (on the sphere with s2, sf's default), not a planar area in squared degrees, and the distortion check passes by construction; with s2 switched off, sf needs the lwgeom package for that area and the request is refused without it. The measured spread is attached as attr("area_error").

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), one column per `agg_funs` entry per variable, `..sd_*` / `..se_*` for every numeric response and predictor, `..neff_*` / `..df_*` / `..ci_lo_*` / `..ci_hi_*` for the same columns when `conf_level` is given, `cell_weight`, `deff_applied` when a design effect was requested (any `deff` other than 1), and `cell_area` / `n_per_area` when `area = TRUE`. An input column also called `n` is not allowed to shadow the count.

`deff_applied` is TRUE on every row when the requested correction was applied (exactly when the "deff_applied" attribute below is attached; under "kish", when any standard-error column was corrected) and FALSE when it fell back to the uncorrected standard errors: a refused `deff`, a "variogram" request with no usable model, or "kish" ICCs of 0 for every variable type (and NA on a `cells_sf` row no point fell in). Unlike the attribute it survives `rbind()` and `dplyr::bind_rows()` of many results. A fallback is also signalled by a warning of class "spatialkit_deff_fallback" (a Kish ICC of 0 is reported on `attr(, "icc")` instead), which `tryCatch(spatialkit_deff_fallback = )` catches without matching the message; it is raised only when the standard errors really are the uncorrected ones.

When a correction was actually applied, an attribute "deff_applied" is attached recording it: method plus `icc_resp/icc_pred` and `deff` for "kish" (`deff` is the primary variable's per-cell design effect, all 1 when only the predictor ICC was positive), `deff/deff_rows/rbar/crs/max_n` for "variogram" (`deff` is the design effect at the primary variable's non-missing count per cell, `deff_rows` at the cell's row count, which is the vector the log line summarises as a median and a max), and `deff` alone for a fixed number. When `cells_sf` is supplied, *every* per-cell vector in that attribute (`deff`, `deff_rows` 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"` request whose ICCs are all 0, or a "variogram" request that could not be fitted. A `deff = "kish"` request always records the ICCs it estimated on an attribute "icc" (`resp` and `pred`, NA for a variable type with no numeric column), whether or not they were positive enough to apply, so a result with no "deff_applied" still says what the ICC came out as.

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. Whole numbers are written out in full for that ("100000", never "1e+05"), so an integer and a double ID of the same cell still match.

Spatial autocorrelation and standard-error bias

By default (`deff = 1`), the `..se_*` columns are computed as `sd / sqrt(n)`, which treats the observations within each cell as independent. When data are spatially autocorrelated (the common

case for the spatial workflows this package supports), within-cell observations are typically positively correlated: they share the cell's departure from the population mean, so as an estimate of the **population (grand) mean** a cell mean has an effective sample size smaller than n . For that estimand the naive SE is **anticonservative** (too small), and a downstream weighted regression that uses `cell_weight` or the `..se_*` columns for population-level inference will produce overconfident standard errors for cells with strong intra-cell correlation. For the cell's **own** mean the naive SE is the right one when the cell's points are spread through it, and the corrected SE is too wide; see "What the standard error estimates" before setting `deff`.

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.

A design effect estimated from the data ("`kish`" or "`variogram`") comes with a second, small-sample correction. The within-cell correlation that inflates the variance of the mean to $\sigma^2 * \text{deff} / n$ also biases the within-cell sample variance downward: under exchangeable correlation ρ (Kish's own assumption), with $\text{deff} = 1 + (n - 1) * \rho$, $E[s^2] = \sigma^2 * (n - \text{deff}) / (n - 1)$, so s^2 understates σ^2 by very nearly the factor by which `deff` inflates the mean's variance, and the two errors compound rather than cancel. The standard error is therefore $s * \sqrt{\text{deff} / n} * \sqrt{(n - 1) / (n - \text{deff})}$, and NA where $\text{deff} \geq n$ (the cell then holds one observation's worth of information and s carries none about σ). This is the package's own derivation, not taken from a reference. Measured 95% interval coverage at $n = 30$ over 20,000 replicates: 0.921, 0.844 and 0.628 at $\rho = 0.2, 0.5$ and 0.8 with $s * \sqrt{\text{deff} / n}$ alone, against 0.948, 0.950 and 0.949 with the rescaling.

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 $\text{sd} * \sqrt{\text{deff} / n}$, exactly $\sqrt{\text{deff}}$ 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 $\text{sd} / \sqrt{n}$ is the better of the two on offer: measured coverage 0.95 under exchangeable within-cell correlation, against very nearly 1.00 for the design-effect-corrected SE, which is too wide by the factor $\sqrt{\text{deff} / (1 - \rho)}$ (4.6 at 20 points a cell and $\rho = 0.5$). 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 right quantity 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, such as a residual one from `estimate_sac_range(..., predictor_vars = )`. A `sac` whose `attr(sac, "detrended")` is TRUE is used as given, but when it corrects response columns a warning says that their standard errors are understated, as `kriging_adequacy()` warns about the same mismatch.

Confidence intervals

With `conf_level` set, every numeric response and predictor column gains four more columns: `..neff_*`, that column's effective sample size in the cell (its non-missing count over its design effect, the per-column version of `cell_weight`); `..df_*`, the degrees of freedom the interval uses; and `..ci_lo_*` / `..ci_hi_*`, a `t` interval for the **cell mean as an estimate of the grand mean**. The estimand is the same as `..se_*`'s, so everything in "What the standard error estimates" applies to it, including that it is not an interval for the cell's own block average. The interval is $\text{mean} \pm \text{qt}((1 + \text{conf_level}) / 2, \text{df}) * \text{se}$, centred on the plain mean of the column's non-missing values whatever `agg_funs` computes, and is NA wherever the standard error is (a single observation; complete redundancy under `deff`). `..neff_*` and `..df_*` are NA where the column has one non-missing value or none, like the standard error; `cell_weight` still counts such a cell (1, or 1 / `deff` for a numeric `deff`), so the two differ there.

The degrees of freedom are **not** `neff - 1`. The interval's spread comes from the within-cell sample variance, and under exchangeable correlation (`deff = "kish"`) that variance keeps its `n - 1` degrees of freedom whatever the design effect. The design effect inflates the mean's variance and biases `s^2`, both of which the standard error already corrects, and the resulting pivot is exactly `t` on `n - 1` df. Measured 95% coverage of the grand mean on the Kish path, 20 cells of 20 at an ICC of 0.2 / 0.6 / 0.9: 0.954 / 0.953 / 0.952 with `n - 1`, against 0.992 / 1.000 / 1.000 with `neff - 1`, which is not an interval so much as a refusal to say anything. The effective-sample-size degrees of freedom of Faes et al. (2009) belong to a mean estimated across many correlated units whose variance is estimated from the spread between them; they do not transfer to a single cell's mean with a variance estimated from inside it. `..df_*` is therefore `n - 1` at `deff = 1`, for a numeric `deff` and for "kish". Under `deff = "variogram"` correlation decays with distance and the within-cell variance loses degrees of freedom to it; `..df_*` is then the Satterthwaite (1946) moment-matched df of `s^2` under the fitted correlation matrix, a fraction of `n - 1` that shrinks as the range grows. Measured on a Gaussian field with an exponential range of 150 and 400 on a 1000-unit domain, 16 cells of about 25 points: 0.960 and 0.958 with that df, against 0.931 and 0.918 with `n - 1`, and 0.34 and 0.19 for the naive `deff = 1`

interval. The interval is only as good as the design effect under it: a mis-specified variogram, or an ICC estimated from too few cells, moves the coverage with it.

References

Faes, C., Molenberghs, G., Aerts, M., Verbeke, G. and Kenward, M. G. (2009). The effective sample size and an alternative small-sample degrees-of-freedom method. *The American Statistician*, 63(4), 389–399. doi:10.1198/tast.2009.08196

Satterthwaite, F. E. (1946). An approximate distribution of estimates of variance components. *Biometrics Bulletin*, 2(6), 110–114. doi:10.2307/3002019

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()`, `kriging_adequacy()`, `resolution_profile()`, `select_resolution()`, `summary.resolution_profile()`

Examples

```
library(sf)
set.seed(1)
n <- 200
east <- runif(n, 0, 100)
north <- runif(n, 0, 100)
# A response with spatial structure, so the within-cell ICC is not zero.
pts <- st_as_sf(
  data.frame(x = 5e5 + east, y = 5e6 + north,
             val = 5 + 0.02 * east + 0.02 * north + rnorm(n, sd = 0.5)),
  coords = c("x", "y"), crs = 32632
)
bnd <- st_sf(geometry = st_sfc(st_polygon(list(rbind(
  c(5e5, 5e6), c(5e5 + 100, 5e6), c(5e5 + 100, 5e6 + 100),
  c(5e5, 5e6 + 100), c(5e5, 5e6)
))), 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. The
# correction is large here, and meant to be: about 22 points per cell that
# all share the cell's part of the trend carry far fewer than 22
# independent pieces of information about the cell mean.
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

# A 95% interval for each cell mean as an estimate of the grand mean, on
# the Kish-corrected standard error and n - 1 degrees of freedom.
```

```
ci <- summarize_by_cell(assigned, response_var = "val", deff = "kish",
                        conf_level = 0.95)
ci[, c("poly_id", "n", "resp_mean_val", "..neff_resp_val", "..df_resp_val",
      "..ci_lo_resp_val", "..ci_hi_resp_val")]
```

summary.resolution_profile

Every criterion's pick, side by side

Description

[select_resolution\(\)](#) reads one criterion at a time. This puts all of them in one table: the level each prefers, the flat region around it, and whether a ladder bound is doing the choosing rather than the criterion. The closing line gives the levels that lie in *every* flat region, the cell counts no criterion objects to.

Usage

```
## S3 method for class 'resolution_profile'
summary(object, criteria = NULL, tol = 0.02, ...)
```

Arguments

object	A resolution_profile() .
criteria	Character vector, any of "cp", "reliability", "elbow", "morán_z". Default: every one of the four that is finite at some level.
tol	Passed to select_resolution() for the flat region. Default 0.02.
...	Ignored.

Details

A flat region is a set, not an interval. The criterion curves are not monotone, so a region can skip a rung of the ladder, and the table prints what the criterion actually accepts ("26, 31", not "26 to 31") rather than a range that would quietly include the levels it rejected.

That set is often empty, and an empty one is a result rather than a failure. The criteria answer different questions: how well the cells represent the field (C_p), whether the cell values are distinguishable from noise (reliability), where the within-cluster sum of squares bends (elbow), and whether the cell means still carry autocorrelation (morán_z). A field with no single right resolution shows up here as disjoint bands, and the spread between the picks is printed for the same reason.

Nothing in the table is a decision procedure. Each flat region is routinely wide, the choice within it belongs to the analyst, and [plot.resolution_profile\(\)](#) draws the curves the bands were read from.

Value

A data.frame of class `resolution_summary`, one row per criterion in the order given, with columns `criterion`, `best`, `flat_min`, `flat_max`, `n_flat`, `value` (the optimum itself, on that criterion's own scale and so not comparable across rows, which is why the print method leaves it out), `at_floor`, `at_ceiling` and `edge` (which bound an edge optimum sits on, as `select_resolution()` reports it; NA when interior). Attributes: `bands` (a named list holding each criterion's flat region in full, since `flat_min` and `flat_max` are only its ends and the region can have holes in it), `common` (the levels in every flat region, an integer vector that is empty when the regions do not overlap), `scored` (the levels each criterion returned a finite value at, which is what decides whether a gap in a band is a rejection or a level nothing was computed at), `ladder` (every level on the profile), `n_levels`, `tol` and `variable` (what the criteria were scored on). The print method recomputes the closing comparison from `bands`, so a row subset of the result reads honestly.

See Also

`select_resolution()` for one criterion, with the per-level values attached; `plot.resolution_profile()` for the curves behind these bands.

Other aggregation: `assign_features_to_polygons()`, `determine_optimal_levels()`, `kriging_adequacy()`, `resolution_profile()`, `select_resolution()`, `summarize_by_cell()`

Examples

```
if (requireNamespace("gstat", quietly = TRUE)) {
  library(sf)
  # An exponential field with range parameter 200 (true effective range
  # 600 m) on a 1 km square, with a nugget of 0.6 on a unit sill: enough
  # noise for Mallows' Cp to have an interior optimum rather than descend
  # to the ceiling.
  set.seed(4)
  n <- 400
  xy <- data.frame(x = 5e5 + runif(n, 0, 1000), y = 5e6 + runif(n, 0, 1000))
  D <- as.matrix(dist(xy))
  xy$z <- as.numeric(t(chol(exp(-D / 200) + diag(0.6, n))) %*% rnorm(n))
  pts <- st_as_sf(xy, coords = c("x", "y"), crs = 32632)
  prof <- resolution_profile(pts, response_var = "z", n_levels = 12)

  # print() is explicit because only the last value of a braced block is
  # shown, and the table is the thing worth seeing here.
  print(summary(prof))
  print(attr(summary(prof), "common")) # the levels all of them accept, if any
  attr(summary(prof), "bands")        # each criterion's region in full
}
```

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

<code>object</code>	A <code>spatial_fit</code> object.
<code>...</code>	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 in place of 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. Neither returns Inf or NaN. Both are averaged over the rows whose denominator is non-zero, and are NA when no row qualifies. Non-zero is judged at the scale of the data: a denominator no larger than 100 machine epsilons times the largest one counts as zero, so the rule does not depend on the units of the response. The `n_MAPE` and `n_SMAPE` columns record how many rows that was; the `n` column counts finite observation/prediction pairs. Read a percentage error next to its count: when `n_MAPE < n`, MAPE is an average over a subset of the data, whatever its value.

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, which `n_MAPE = 58` now says. 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; `n_SMAPE` shows how many rows it kept, and the count is only a label, not a repair.

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 methods on a fitted model: `coef.bayesian_fit()`, `coef.gwr_fit()`, `coef.rf_fit()`, `fitted.bayesian_fit()`, `fitted.gwr_fit()`, `fitted.rf_fit()`, `model_metrics()`, `predict.bayesian_fit()`, `predict.gwr_fit()`, `predict.rf_fit()`, `print.rf_fit()`, `print.spatial_fit()`, `residuals.bayesian_fit()`, `residuals.gwr_fit()`, `residuals.rf_fit()`

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 follow the data, so that counts per cell vary far less than on a fixed grid over clustered points (k -means does not equalise them, it minimises the spread of points around each centre), which is what keeps per-cell aggregates in `summarize_by_cell()` from resting on one or two observations. 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, nstart = 10)
```

Arguments

<code>points_sf</code>	An sf object with POINT geometries.
<code>k</code>	Integer; requested number of clusters, treated as an upper bound only. It is clamped, with a warning, to whichever is smaller of the number of distinct point positions and <code>nrow(points_sf) - 1</code> , because k -means can produce neither more centres than there are distinct points nor as many centres as there are rows. Check <code>nrow()</code> on the result. A <code>select_resolution()</code> result or a <code>resolution_profile()</code> is also accepted: its scored centres are returned when it carries them, and otherwise its count is used.
<code>set_seed</code>	Optional integer RNG seed. Default 456, so a call gives the same seeds every time whatever the session's random-number state; an outer <code>set.seed()</code> does not change them, and the caller's random-number stream is left as it was. Pass NULL to draw the k -means starts from the session's stream instead (the default of <code>get_voronoi_seeds()</code>).
<code>nstart</code>	Number of random starts for <code>stats::kmeans()</code> ; the best is kept. Default 10.

Details

Lon/lat input is projected first so the k-means distances are metric and not degrees; so is input with no CRS whose coordinates look like lon/lat (the heuristic [ensure_projected\(\)](#) applies, with its warning), and the seeds come back in the input's own coordinates. Rows with empty or non-finite coordinates are dropped with a warning, and k is clamped to the number of distinct positions.

The partition is [stats::kmeans\(\)](#) (Hartigan-Wong) with `nstart` random starts. [resolution_profile\(\)](#) and [determine_optimal_levels\(\)](#) score each count on a different run, by default the best of 25 k-means++ restarts, which usually reaches a lower within-cluster sum of squares; the seeds for a count passed as a number are therefore not the partition that count was scored on. Raising `nstart` narrows the gap but does not close it. Pass the [select_resolution\(\)](#) result or the profile itself as `k` to close it: the centres of the scored partition are then returned (in the CRS of `points_sf`, with `attr(, "n_from")`), and no k-means is run.

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\(\)](#).

See Also

Other tessellation: [build_tessellation\(\)](#), [create_grid_polygons\(\)](#), [create_grid_polygons_cached\(\)](#), [create_voronoi_polygons\(\)](#), [ensure_stable_poly_id\(\)](#), [get_voronoi_seeds\(\)](#), [plot_tessellation_map\(\)](#), [voronoi_seeds_random\(\)](#)

Examples

```
library(sf)
set.seed(1)
pts <- st_as_sf(
  data.frame(x = 5e5 + runif(100, 0, 1000), y = 5e6 + runif(100, 0, 1000)),
  coords = c("x", "y"), crs = 32632
)
seeds <- voronoi_seeds_kmeans(pts, k = 8)
nrow(seeds) # at most 8: one seed per non-empty cluster
seeds      # the cluster centres, as an sf POINT layer in the points' CRS
```

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) instead of 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 = NULL)
```

Arguments

boundary	An sf or sfc polygonal object.
k	Integer; number of random seeds.
set_seed	Optional integer RNG seed. Default NULL: the seeds are drawn from the session's random-number stream, so consecutive calls give different seedings and set.seed() before a call makes it reproducible. Pass a number to get the same seeds whatever that stream holds; the caller's stream is then left as it was. The default used to be 456, which made every call return the same "random" seeding, even inside a loop over set.seed() .

Details

Seeds are drawn uniformly inside the polygon. When a draw falls short of k it is topped up with further uniform draws from the same polygon, so the result has exactly k seeds; only a geometry that still yields too few after ten top-ups returns fewer, and that shortfall is logged.

Value

An sf object of k random POINTs (fewer only in the degenerate case above), with seed_id and method = "random" columns matching [get_voronoi_seeds\(\)](#).

See Also

Other tessellation: [build_tessellation\(\)](#), [create_grid_polygons\(\)](#), [create_grid_polygons_cached\(\)](#), [create_voronoi_polygons\(\)](#), [ensure_stable_poly_id\(\)](#), [get_voronoi_seeds\(\)](#), [plot_tessellation_map\(\)](#), [voronoi_seeds_kmeans\(\)](#)

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))
set.seed(1)
seeds <- voronoi_seeds_random(bnd, k = 10)
nrow(seeds) # 10
seeds
# Another call is another seeding; set_seed pins one.
identical(st_coordinates(voronoi_seeds_random(bnd, k = 10, set_seed = 7)),
  st_coordinates(voronoi_seeds_random(bnd, k = 10, set_seed = 7)))
```

Index

- * **aggregation**
 - assign_features_to_polygons, [10](#)
 - determine_optimal_levels, [50](#)
 - kriging_adequacy, [93](#)
 - resolution_profile, [143](#)
 - select_resolution, [153](#)
 - summarize_by_cell, [155](#)
 - summary.resolution_profile, [163](#)
- * **cross-validation**
 - area_of_applicability, [5](#)
 - cv_bayes, [34](#)
 - cv_block_size_sweep, [38](#)
 - cv_gwr, [40](#)
 - cv_rf, [43](#)
 - cv_spatial, [45](#)
 - estimate_sac_range, [60](#)
 - fold_separation, [84](#)
 - gwr_model_selection, [89](#)
 - make_folds, [97](#)
 - sac_nugget, [148](#)
 - select_features_forward, [149](#)
- * **methods on a fitted model**
 - coef.bayesian_fit, [18](#)
 - coef.gwr_fit, [19](#)
 - coef.rf_fit, [20](#)
 - fitted.bayesian_fit, [81](#)
 - fitted.gwr_fit, [83](#)
 - fitted.rf_fit, [83](#)
 - model_metrics, [105](#)
 - predict.bayesian_fit, [125](#)
 - predict.gwr_fit, [126](#)
 - predict.rf_fit, [127](#)
 - print.rf_fit, [135](#)
 - print.spatial_fit, [137](#)
 - residuals.bayesian_fit, [141](#)
 - residuals.gwr_fit, [142](#)
 - residuals.rf_fit, [143](#)
 - summary.spatial_fit, [164](#)
- * **model evaluation**
 - compare_models, [22](#)
 - compare_models_cv, [24](#)
 - evaluate_insample, [67](#)
 - model_metrics, [105](#)
 - residual_morans_i, [137](#)
- * **model fitting**
 - fit_bayesian_spatial_model, [69](#)
 - fit_gwr_model, [75](#)
 - fit_rf_model, [78](#)
 - gp_lengthscales_bounds, [87](#)
 - new_spatial_fit, [107](#)
 - prep_model_data, [131](#)
- * **package options and caches**
 - clear_fitted_cache, [15](#)
 - clear_grid_cache, [16](#)
 - create_grid_polygons_cached, [30](#)
 - spatialkit_quiet, [154](#)
- * **plotting**
 - plot.aoa, [109](#)
 - plot.block_size_sweep, [111](#)
 - plot.feature_selection, [112](#)
 - plot.gwr_model_selection, [113](#)
 - plot.resolution_profile, [114](#)
 - plot.sac_range, [115](#)
 - plot.spatial_fit, [116](#)
 - plot_calibration, [119](#)
 - plot_cv_metrics, [120](#)
 - plot_folds, [121](#)
- * **prediction**
 - area_of_applicability, [5](#)
 - predict_surface, [128](#)
- * **print methods**
 - print.aoa, [133](#)
 - print.gwr_model_selection, [133](#)
 - print.morans_i, [134](#)
 - print.resolution_profile, [135](#)
 - print.sac_range, [136](#)
- * **spatial data preparation**
 - clip_target_for, [17](#)

- coerce_to_points, 21
- ensure_projected, 55
- harmonize_crs, 92
- prep_model_data, 131
- * tessellation**
 - build_tessellation, 12
 - create_grid_polygons, 28
 - create_grid_polygons_cached, 30
 - create_voronoi_polygons, 32
 - ensure_stable_poly_id, 58
 - get_voronoi_seeds, 86
 - plot_tessellation_map, 122
 - voronoi_seeds_kmeans, 166
 - voronoi_seeds_random, 167
- [.spatialkit_rows, 4, 11, 132
- area_of_applicability, 5, 37, 40, 42, 45, 49, 78, 81, 85, 91, 104, 109, 110, 130, 149, 152
- area_of_applicability(), 66
- assign_features_to_polygons, 4, 5, 10, 12, 55, 84, 94, 97, 154, 162, 164
- assign_features_to_polygons(), 55, 148, 156, 158, 162
- build_tessellation, 12, 30, 31, 33, 87, 101, 103, 124, 148, 167, 168
- build_tessellation(), 11, 18, 55, 59, 162
- clear_fitted_cache, 15, 16, 31, 82, 155
- clear_grid_cache, 16, 16, 31, 155
- clip_target_for, 17, 22, 57, 93, 132
- clip_target_for(), 12
- coef, 108
- coef.bayesian_fit, 18, 20, 21, 71, 74, 82–84, 107, 126–128, 136, 137, 142, 143, 166
- coef.gwr_fit, 19, 19, 21, 82–84, 107, 118, 126–128, 136, 137, 142, 143, 166
- coef.rf_fit, 19, 20, 20, 82–84, 107, 126–128, 136, 137, 142, 143, 166
- coerce_to_points, 9, 18, 21, 57, 93, 129, 131, 132
- compare_models, 22, 23, 27, 67, 68, 70, 71, 106, 107, 141
- compare_models_cv, 23, 24, 42, 67, 68, 80, 105–107, 119, 120, 141, 165
- create_grid_polygons, 15, 28, 31, 33, 87, 124, 129, 167, 168
- create_grid_polygons(), 14, 17, 31, 32, 59
- create_grid_polygons_cached, 15, 16, 30, 30, 33, 87, 124, 155, 167, 168
- create_grid_polygons_cached(), 59
- create_voronoi_polygons, 14, 15, 30, 31, 32, 87, 124, 167, 168
- create_voronoi_polygons(), 17, 28, 59
- cv_bayes, 9, 23, 27, 34, 36, 37, 40, 42, 45, 49, 68, 69, 71, 73, 85, 91, 104–106, 119, 120, 149, 152, 165
- cv_bayes(), 66
- cv_block_size_sweep, 9, 37, 38, 42, 45, 49, 85, 91, 104, 111, 149, 152
- cv_block_size_sweep(), 66
- cv_gwr, 9, 25, 37, 40, 40, 45, 47, 49, 83, 85, 91, 104, 105, 120, 149, 152
- cv_gwr(), 66
- cv_rf, 9, 25, 36, 37, 40, 42, 43, 47, 49, 80, 81, 85, 91, 104, 105, 120, 143, 149, 152
- cv_rf(), 66
- cv_spatial, 8, 9, 26, 35–40, 42–44, 45, 45, 78, 80, 85, 91, 104, 107, 108, 120, 149, 152
- cv_spatial(), 66
- determine_optimal_levels, 11, 13, 14, 50, 97, 145–148, 151, 154, 162, 164
- determine_optimal_levels(), 86, 148, 167
- ensure_projected, 18, 21, 22, 55, 76, 84, 93, 99, 100, 103, 132
- ensure_projected(), 13, 14, 29, 32, 33, 87, 92, 158, 167
- ensure_stable_poly_id, 15, 30, 31, 33, 58, 87, 124, 167, 168
- estimate_sac_range, 9, 37, 39, 40, 42, 45, 49, 60, 73, 84, 85, 91, 96, 99, 101, 104, 115, 117, 145, 148, 149, 152
- estimate_sac_range(), 157
- evaluate_insample, 22, 23, 27, 67, 67, 106, 107, 141
- fit_bayesian_spatial_model, 23, 34, 36, 69, 78, 81, 88, 107, 108, 132, 160
- fit_gwr_model, 41, 71, 74, 75, 81, 88–91, 107, 108, 132, 150
- fit_rf_model, 23, 25, 44, 67, 71, 74, 78, 78, 83, 88, 105, 107, 108, 127, 132, 150, 165

- `fitted.bayesian_fit`, [15](#), [19–21](#), [81](#), [83](#), [84](#),
[107](#), [126–128](#), [136](#), [137](#), [141–143](#),
[166](#)
- `fitted.gwr_fit`, [19–21](#), [82](#), [83](#), [84](#), [107](#),
[126–128](#), [136](#), [137](#), [142](#), [143](#), [166](#)
- `fitted.rf_fit`, [19–21](#), [82](#), [83](#), [83](#), [107](#),
[126–128](#), [136](#), [137](#), [142](#), [143](#), [166](#)
- `fold_separation`, [9](#), [37](#), [40](#), [42](#), [45](#), [48](#), [49](#),
[84](#), [91](#), [104](#), [149](#), [152](#)
- `fold_separation()`, [66](#)
- `get_voronoi_seeds`, [13](#), [15](#), [30](#), [31](#), [33](#), [86](#),
[103](#), [124](#), [146](#), [148](#), [153](#), [167](#), [168](#)
- `get_voronoi_seeds()`, [55](#), [59](#), [166–168](#)
- `gp_lengthscales_bounds`, [74](#), [78](#), [81](#), [87](#), [108](#),
[132](#)
- `gwr_model_selection`, [9](#), [37](#), [40](#), [42](#), [45](#), [49](#),
[85](#), [89](#), [104](#), [113](#), [149](#), [152](#)
- `gwr_model_selection()`, [66](#)
- `harmonize_crs`, [18](#), [22](#), [57](#), [92](#), [132](#)
- `harmonize_crs()`, [13](#), [17](#), [32](#)
- `kriging_adequacy`, [11](#), [55](#), [93](#), [154](#), [162](#), [164](#)
- `kriging_adequacy()`, [148](#), [161](#)
- `make_folds`, [6](#), [7](#), [9](#), [25](#), [34](#), [37](#), [40–42](#), [44–46](#),
[49](#), [54](#), [62](#), [66](#), [84](#), [85](#), [91](#), [94](#), [95](#), [97](#),
[149](#), [151](#), [152](#)
- `make_folds()`, [21](#), [33](#), [57](#), [66](#)
- `model_metrics`, [19–21](#), [23](#), [27](#), [40](#), [48](#), [68](#), [73](#),
[82–84](#), [105](#), [108](#), [126–128](#), [136](#), [137](#),
[141–143](#), [166](#)
- `model_metrics()`, [78](#)
- `new_spatial_fit`, [19](#), [20](#), [46](#), [49](#), [74](#), [78](#), [81](#),
[88](#), [107](#), [117](#), [132](#), [139](#)
- `plot.aoa`, [109](#), [111–113](#), [115](#), [116](#), [118](#), [119](#),
[121](#), [122](#)
- `plot.block_size_sweep`, [40](#), [110](#), [111](#), [112](#),
[113](#), [115](#), [116](#), [118](#), [119](#), [121](#), [122](#)
- `plot.feature_selection`, [110](#), [111](#), [112](#),
[113](#), [115](#), [116](#), [118](#), [119](#), [121](#), [122](#),
[151](#)
- `plot.gwr_model_selection`, [110–112](#), [113](#),
[115](#), [116](#), [118](#), [119](#), [121](#), [122](#)
- `plot.resolution_profile`, [110–113](#), [114](#),
[116](#), [118](#), [119](#), [121](#), [122](#), [163](#), [164](#)
- `plot.sac_range`, [66](#), [110–113](#), [115](#), [115](#), [118](#),
[119](#), [121](#), [122](#)
- `plot.spatial_fit`, [19](#), [110–113](#), [115](#), [116](#),
[116](#), [119](#), [121](#), [122](#)
- `plot_calibration`, [110–113](#), [115](#), [116](#), [118](#),
[119](#), [121](#), [122](#)
- `plot_cv_metrics`, [110–113](#), [115](#), [116](#), [118](#),
[119](#), [120](#), [122](#)
- `plot_folds`, [103](#), [110–113](#), [115](#), [116](#), [118](#),
[119](#), [121](#), [121](#)
- `plot_tessellation_map`, [15](#), [30](#), [31](#), [33](#), [87](#),
[122](#), [167](#), [168](#)
- `plot_tessellation_map()`, [59](#)
- `predict.bayesian_fit`, [19–21](#), [82–84](#), [107](#),
[125](#), [127](#), [128](#), [130](#), [136](#), [137](#), [142](#),
[143](#), [166](#)
- `predict.gwr_fit`, [19–21](#), [82–84](#), [107](#), [126](#),
[126](#), [128](#), [136](#), [137](#), [142](#), [143](#), [166](#)
- `predict.rf_fit`, [19–21](#), [82–84](#), [107](#), [126](#),
[127](#), [127](#), [136](#), [137](#), [142](#), [143](#), [166](#)
- `predict_surface`, [5](#), [9](#), [78](#), [100](#), [126](#), [128](#), [128](#)
- `prep_model_data`, [4](#), [5](#), [18](#), [22](#), [57](#), [74](#), [78](#), [81](#),
[88](#), [93](#), [108](#), [131](#)
- `prep_model_data()`, [21](#), [33](#)
- `print.aoa`, [133](#), [134–136](#)
- `print.gwr_model_selection`, [133](#), [133–136](#)
- `print.morans_i`, [133](#), [134](#), [134–136](#)
- `print.resolution_profile`, [133](#), [134](#), [135](#),
[136](#)
- `print.rf_fit`, [19–21](#), [82–84](#), [107](#), [126–128](#),
[135](#), [137](#), [142](#), [143](#), [166](#)
- `print.sac_range`, [133–135](#), [136](#)
- `print.spatial_fit`, [19–21](#), [82–84](#), [107](#),
[126–128](#), [136](#), [137](#), [142](#), [143](#), [166](#)
- `residual_morans_i`, [23](#), [27](#), [68](#), [107](#), [134](#),
[137](#), [143](#)
- `residuals`, [108](#)
- `residuals.bayesian_fit`, [19–21](#), [82–84](#),
[107](#), [126–128](#), [136](#), [137](#), [141](#), [142](#),
[143](#), [166](#)
- `residuals.gwr_fit`, [19–21](#), [82–84](#), [107](#),
[126–128](#), [136](#), [137](#), [142](#), [142](#), [143](#),
[166](#)
- `residuals.rf_fit`, [19–21](#), [82–84](#), [107](#),
[126–128](#), [136](#), [137](#), [142](#), [143](#), [166](#)
- `resolution_profile`, [11](#), [13](#), [53–55](#), [64](#), [97](#),
[114](#), [143](#), [153](#), [154](#), [162–164](#)
- `resolution_profile()`, [86](#), [87](#), [166](#), [167](#)

sac_nugget, [9](#), [37](#), [40](#), [43](#), [45](#), [49](#), [64](#), [66](#), [85](#),
[91](#), [104](#), [148](#), [152](#)
sac_nugget(), [66](#)
select_features_forward, [9](#), [37](#), [40](#), [43](#), [45](#),
[49](#), [85](#), [90](#), [91](#), [104](#), [112](#), [149](#), [149](#)
select_features_forward(), [66](#)
select_resolution, [11](#), [13](#), [55](#), [97](#), [114](#), [143](#),
[148](#), [153](#), [162–164](#)
select_resolution(), [86](#), [148](#), [166](#), [167](#)
set.seed(), [166](#), [168](#)
sf::sf_use_s2(), [56](#)
sf::st_buffer(), [17](#)
sf::st_crs(), [14](#), [29](#), [33](#), [56](#), [92](#)
sf::st_join(), [11](#)
sf::st_line_sample(), [21](#)
sf::st_make_grid(), [29](#)
sf::st_set_crs(), [92](#)
sf::st_voronoi(), [33](#)
spatialkit_quiet, [14](#), [16](#), [17](#), [25](#), [29](#), [31](#), [33](#),
[71](#), [151](#), [154](#), [158](#)
spatialkit_quiet(), [56](#)
stats::kmeans(), [87](#), [166](#), [167](#)
summarize_by_cell, [11](#), [12](#), [55](#), [93](#), [94](#), [97](#),
[124](#), [154](#), [155](#), [164](#)
summarize_by_cell(), [11](#), [55](#), [56](#), [148](#), [166](#)
summary, [108](#), [137](#)
summary.resolution_profile, [11](#), [55](#), [97](#),
[154](#), [162](#), [163](#)
summary.resolution_profile(), [148](#)
summary.spatial_fit, [19–21](#), [82–84](#), [107](#),
[126–128](#), [136](#), [137](#), [142](#), [143](#), [164](#)

voronoi_seeds_kmeans, [15](#), [30](#), [31](#), [33](#), [87](#),
[124](#), [146](#), [153](#), [166](#), [168](#)
voronoi_seeds_kmeans(), [59](#), [167](#)
voronoi_seeds_random, [15](#), [30](#), [31](#), [33](#), [87](#),
[124](#), [167](#), [167](#)
voronoi_seeds_random(), [59](#), [166](#)