Package: soilcnn (via r-universe)

October 21, 2026

Title Convolutional Neural Networks for Digital Soil Mapping

Version 0.1.0

Description A framework, built on 'torch', for mapping a soil property from point observations and a stack of aligned rasters with a convolutional network that reads the neighbourhood of each point. It extracts the patches; resamples by spatial blocks or by k-fold nearest neighbour distance matching, after Linnenbrink et al. (2024) <doi:10.5194/gmd-17-5897-2024>; tunes and selects under a noise floor and the one-standard-error rule; refits the selected configuration under several seeds; and predicts median, mean and interval maps over rasters of any extent. The map's centre is the ensemble median over seeds; the mean corrects its back-transformation by smearing, after Duan (1983) <doi:10.1080/01621459.1983.10478017>. The intervals are conformal, calibrated three ways and each checked on the test set: on the cross-validated residuals; on a calibration set held out of every fold, as split conformal, after Lei et al. (2018) <doi:10.1080/01621459.2017.1307116>; or with every fold's model at each pixel, as CV+, after Barber et al. (2021) <doi:10.1214/20-AOS1965> -- with points, or whole groups of them, weighing alike, after Dunn et al. (2023) <doi:10.1080/01621459.2022.2060112>. The area of applicability of Meyer and Pebesma (2021) <doi:10.1111/2041-210X.13650> is mapped beside them. What a model learned is read by permutation, by SHAP values from expected gradients, after Erion et al. (2021) <doi:10.1038/s42256-021-00343-w>, by SAGE, after Covert et al. (2020) <doi:10.48550/arXiv.2004.00668>, by accumulated local effects, after Apley and Zhu (2020) <doi:10.1111/rssb.12377>, and by refits without each variable. Random forest and multilayer perceptron baselines run under the same folds.

License MIT + file LICENSE

URL https://github.com/moquedace/soilcnn, https://moquedace.github.io/soilcnn/

BugReports https://github.com/moquedace/soilcnn/issues

Depends R (>= 4.5.0)

Imports callr, coro, dplyr, graphics, grDevices, janitor, matrixStats, parallel, ps, purrr, readr, stats, terra, tibble, tools, torch, utils

Suggests caret, CAST, FNN, knitr, pkgload, randomForest, ranger, rmarkdown, sf, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

Encoding UTF-8

Language en-GB

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Cássio Marques Moquedace [aut, cre, cph], Clara Glória Oliveira Baldi [aut, cph]

Maintainer Cássio Marques Moquedace <cassiomoquedace@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-10-21 10:19:38 UTC

RemoteUrl https://github.com/cran/soilcnn

RemoteRef HEAD

RemoteSha 2379c381d2cc1dea08270023c35fab0d5565455e

Contents

ale_effect 4
aoa_di 5
aoa_reference 6
aoa_threshold 7
block_bootstrap 8
block_bootstrap_by_size 9
calc_metrics 10
caret_available 11
caret_spec 11
ccc 12
check_fold_plan 13
compare_importance 13
conformal_calibrate 14
conformal_cv 15
conformal_interval 16
conformal_scale_fit 17
conformal_scaled_calibrate 17
conformal_scaled_interval 19
context_importance 20

cv_plus_calibrate . . . 21
cv_plus_interval . . . 22
cv_residuals . . . 22
cv_residuals_for_config . . . 23
di_reference . . . 24
dissimilarity_index . . . 25
dsm_final . . . 25
dsm_importance . . . 27
dsm_load . . . 30
dsm_predict . . . 31
dsm_prepare . . . 33
dsm_report_final . . . 36
dsm_train . . . 37
equal_area_blocks . . . 39
example_landscape . . . 40
example_run . . . 41
fold_leakage_report . . . 41
freeze_selection . . . 42
get_model . . . 43
holdout . . . 43
holdout_cv . . . 45
importance_groups . . . 46
importance_map . . . 47
importance_points . . . 48
importance_weights . . . 49
inside_aoa . . . 49
knndm_cv . . . 50
knndm_folds . . . 51
latest_run_dir . . . 53
list_models . . . 54
make_manual_tune_grid . . . 54
make_tune_grid . . . 55
model_spec . . . 56
occlusion_report . . . 57
one_se . . . 59
paired_family_test . . . 60
permutation_importance . . . 61
picp_report . . . 62
plot.block_bootstrap_by_size . . . 63
plot.dsm_importance . . . 64
plot.importance_comparison . . . 64
plot.importance_map . . . 65
plot.spatial_correlogram . . . 65
prediction_sample . . . 66
print_aoa . . . 66
print_noise_floor . . . 67
print_one_se . . . 68
random_cv . . . 68

random_folds 69
refit_importance 71
region_cv 72
region_folds 73
register_model 74
resolve_resampling 75
sage_importance 76
score_test_grid 77
seed_noise_floor 78
selected_config_id 79
set_torch_threads 79
setup_torch_device 80
shap_importance 81
smear 82
smear_map 83
smearing_check 84
smearing_factor 85
smearing_from_run 86
spatial_correlogram 87
spatial_cv 88
spatial_folds 90
summarise_resamples 91
target_transform_spec 92

Index **93**

`ale_effect` *ALE: how the prediction changes along each variable's range.*

Description

Accumulated local effects (Apley & Zhu 2020). A variable's range is cut into bins by its quantiles at the points; within each bin, every point is moved to the bin's lower edge and to its upper edge, and the difference of the two predictions is the local effect. Accumulated across the bins and centred, it is the curve. A point is moved only across its own bin, so the model is asked about values close to the ones it saw – which is why ALE is read with correlated predictors, where a partial dependence plot would ask about combinations that do not occur.

Usage

```
ale_effect(variables = NULL, bins = 20L)
```

Arguments

`variables`	Which variables, named as `importance_groups()` names them (a channel, or a one-hot set). NULL: all of them. Groups of your own are not used: an effect needs one variable's values.
`bins`	Bins of a continuous variable's range, cut at its quantiles at the points – fewer where it has fewer distinct values.

Details

A point's whole patch is moved, every pixel and every window by the same amount: the neighbourhood keeps its texture and changes its level. Moving the centre pixel alone would build a spike no landscape has. A categorical – a one-hot set, or a binary map alone – has no range: its effect is the prediction with the whole patch made each class, less the prediction as it is. That is a partial dependence, which asks about a class also where it does not occur; the number of points in each class says where it does.

Value

An `importance_spec`, for `dsm_importance()`. Its importance is the spread of the curve over the points – the standard deviation of the effect at their values; flat is no effect (Greenwell et al. 2018 for the idea, on partial dependence).

Examples

```
ale_effect(bins = 10)
```

---

aoa_di — *The DI of new rows of RAW predictor values, QC'd and scaled as the reference was.*

Description

The DI of new rows of RAW predictor values, QC'd and scaled as the reference was.

Usage

```
aoa_di(aref, values)
```

Arguments

`aref`	From aoa_reference().
`values`	Matrix or data frame of raw values, columns in the model's order.

Value

The DI of each row; NA where a channel is missing after QC.

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
                     percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                     overwrite = TRUE, verbose = FALSE)
data <- dsm_load(store, windows = integer(0), verbose = FALSE)
predictors <- as.character(data$store$predictors)
```

```
qc <- utils::read.csv2(file.path(data$patch_dir, "qc_table.csv"))
qc <- qc[match(predictors, qc$predictor), ]
# a fitted model's scaling is its predictor_scaling.csv; here, the profiles' own
x <- as.matrix(data$points[, predictors])
scaling <- data.frame(predictor = predictors, center = colMeans(x),
                      scale = apply(x, 2, sd))
plan <- spatial_folds(data$store$meta, k = 3, block_size = 0.05)
aref <- aoa_reference(data$points, predictors, qc, scaling, plan)
aoa_di(aref, data$points[1:5, predictors])
```

`aoa_reference` *The dissimilarity reference of a fitted model.*

Description

The centre pixel of every point that trained or validated in the tuning plan, QC'd and scaled with the MODEL's scaling (stage 07's construction). Each point's fold is the one it was held out in, so its cross-validated DI is the distance to the nearest point OUTSIDE its fold: the dissimilarity the model that predicted it during cross-validation actually faced. That is the DI a calibration residual comes with. A point that only ever trained – the training side of a holdout – has fold `NA` and no cross-validated DI: it is in the reference, a neighbour to every fold, and out of the threshold.

Usage

```
aoa_reference(points, predictors, qc_table, scaling, plan, weights = NULL)
```

Arguments

`points`	Point table aligned to the store (align_points_to_meta()).
`predictors`	Channel names, in the model's order.
`qc_table`	QC rules, in the same order.
`scaling`	The fitted model's predictor_scaling, in the same order.
`plan`	The tuning run's fold plan.
`weights`	NULL (every channel alike), or one non-negative weight per channel, in the same order – an importance, from importance_weights(), so a point is unlike the training data in what the model uses (Meyer & Pebesma 2021).

Value

An `aoa_reference`: the DI reference, the AOA threshold, and each used point's fold and cross-validated DI (`NA` for a point never held out).

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
                     percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                     overwrite = TRUE, verbose = FALSE)
data <- dsm_load(store, windows = integer(0), verbose = FALSE)
predictors <- as.character(data$store$predictors)
qc <- utils::read.csv2(file.path(data$patch_dir, "qc_table.csv"))
qc <- qc[match(predictors, qc$predictor), ]
# a fitted model's scaling is its predictor_scaling.csv; here, the profiles' own
x <- as.matrix(data$points[, predictors])
scaling <- data.frame(predictor = predictors, center = colMeans(x),
                      scale = apply(x, 2, sd))
plan <- spatial_folds(data$store$meta, k = 3, block_size = 0.05)
aref <- aoa_reference(data$points, predictors, qc, scaling, plan)
as.numeric(aref$threshold)     # beyond it, the cross-validated error does not apply
head(aref$cv)
```

`aoa_threshold` *The DI threshold that separates the area of applicability.*

Description

Derived from the CROSS-VALIDATED training data: for each training row, the distance to the nearest training row in a DIFFERENT fold. That set is the dissimilarity the model actually coped with during cross-validation, so the outlier-removed maximum of it is the largest dissimilarity for which the reported error has been demonstrated.

Usage

```
aoa_threshold(ref, folds, k_iqr = 1.5)
```

Arguments

`ref`	From di_reference().
`folds`	Integer fold label per training row, in the same order as the matrix the reference was built from; `NA` for a row never held out.
`k_iqr`	Outlier rule: threshold = Q75 + k_iqr * IQR. 1.5 is Tukey's fence and the value Meyer & Pebesma use.

Details

A row never held out – the training side of a single split – has fold `NA`: it is a neighbour to every fold, and has no cross-validated DI of its own, since no model predicted it.

Value

The threshold, with the cross-validated DI attached as "cv_di" (`NA` for the rows never held out).

Examples

```
set.seed(1)
x_train <- matrix(rnorm(200), 100, 2)    # 100 training rows of 2 scaled predictors
ref <- di_reference(x_train)
th <- aoa_threshold(ref, folds = rep(1:5, 20))
as.numeric(th)                 # the threshold
summary(attr(th, "cv_di"))     # the cross-validated DI it was taken from
```

`block_bootstrap` *A metric, or two models' difference in it, with an interval from whole blocks.*

Description

The points are drawn by blocks: each draw takes as many blocks as there are, with replacement, and every point of a block drawn comes with it. With `against`, the statistic is the difference `pred` less `against`, at the same points. Two weightings answer two questions: "profile", every point one vote; "block", every block one vote, its points weighing 1/n each. They differ in the estimate as well as in the interval, and where they disagree the advantage rests on the densely sampled blocks.

Usage

```
block_bootstrap(
  obs,
  pred,
  blocks,
  against = NULL,
  metric = c("mae", "rmse", "ccc", "bias"),
  weights = c("profile", "block"),
  n_boot = 2000L,
  conf = 0.95,
  seed = 42L
)
```

Arguments

`obs`	Observations.
`pred`	Predictions at the same points.
`blocks`	One block per point: from `equal_area_blocks()`, or the blocks the points were drawn by.
`against`	NULL, or another model's predictions at the same points.
`metric`	One or more of "mae", "rmse", "ccc" and "bias" (the mean of pred - obs).
`weights`	"profile" and/or "block".
`n_boot`	Draws of the blocks.
`conf`	The interval's level.
`seed`	Seed of the draws.

Value

A `block_bootstrap`: a tibble, one row per metric and weighting, with `estimate` (pred's), `against` (the other model's), `difference`, `ci_low` and `ci_high` (of the difference when `against` is given, of the estimate otherwise), `n_points`, `n_blocks`, `n_effective` (for "profile": the number of independent points that would give the same interval) and `share_better` (the share of points, or of blocks, where `pred` errs less).

Examples

```
ex <- example_landscape()
obs <- ex$profiles$soc_stock
set.seed(1)
pred_a <- obs * exp(rnorm(length(obs), 0, 0.20))   # two models' predictions
pred_b <- obs * exp(rnorm(length(obs), 0, 0.25))
blk <- equal_area_blocks(ex$profiles$x, ex$profiles$y, size_km = 2)
block_bootstrap(obs, pred_a, blk, against = pred_b, metric = "mae", n_boot = 500)
```

---

block_bootstrap_by_size

The interval as the blocks grow.

Description

`block_bootstrap()` for blocks of growing side. Blocks smaller than the distance over which the values resemble each other are themselves dependent, and their interval is still too narrow; where the width stops growing, they no longer are.

Usage

```
block_bootstrap_by_size(
  x,
  y,
  obs,
  pred,
  against = NULL,
  sizes_km = c(25, 50, 100, 200, 400),
  metric = "mae",
  weights = "profile",
  n_boot = 2000L,
  conf = 0.95,
  seed = 42L,
  coords = c("lonlat", "metres"),
  centre = NULL
)
```

Arguments

x, y	Coordinates, as in equal_area_blocks().
obs, pred, against	As in block_bootstrap().
sizes_km	The block sides to try, in km.
metric	One metric, as in block_bootstrap().
weights	One weighting, as in block_bootstrap().
n_boot, conf, seed	As in block_bootstrap().
coords, centre	As in equal_area_blocks().

Value

A block_bootstrap_by_size: a tibble, one row per size, with size_km, n_blocks, the estimate (or the difference), ci_low, ci_high, width and n_effective.

Examples

```
ex <- example_landscape()
obs <- ex$profiles$soc_stock
set.seed(1)
pred_a <- obs * exp(rnorm(length(obs), 0, 0.20))   # two models' predictions
pred_b <- obs * exp(rnorm(length(obs), 0, 0.25))
s <- block_bootstrap_by_size(ex$profiles$x, ex$profiles$y, obs, pred_a, against = pred_b,
                             sizes_km = c(1, 2, 5, 10), n_boot = 500)
s
plot(s)
```

---

calc_metrics *Compute all regression metrics for one obs/pred pair.*

Description

Compute all regression metrics for one obs/pred pair.

Usage

```
calc_metrics(obs, pred)
```

Arguments

obs	Numeric vector of observed values.
pred	Numeric vector of predicted values (same length as obs).

Value

A one-row tibble with columns: n, ccc, r2, mae, nse, rmse, rpd, mqi, bias (signed, native units) and bias_pct (relative to mean(obs)).

Examples

```
obs  <- c(12, 25, 31, 48, 60)
pred <- c(15, 22, 35, 40, 66)
calc_metrics(obs, pred)
```

---

caret_available — *Every caret method that could be borrowed here.*

Description

Every caret method that could be borrowed here.

Usage

```
caret_available(pattern = NULL)
```

Arguments

pattern	Optional regular expression on the method name.

Value

A tibble of method, label and the parameters it tunes.

Examples

```
caret_available("^rf$|^ranger$")
```

---

caret_spec — *Turn a caret method into a model_spec.*

Description

Turn a caret method into a model_spec.

Usage

```
caret_spec(method, name = method, search = c("grid", "random"), ...)
```

Arguments

`method`	caret method name, e.g. "ranger", "xgbTree", "cubist".
`name`	Name to register it under. Defaults to the method name.
`search`	Passed to caret_grid() for the default grid.
`...`	Extra arguments forwarded to every caret::train() call – this is where a model's own arguments go (num.threads, nthread, ...).

Value

A model_spec with input == "table".

Examples

```
caret_spec("rf")
```

---

ccc *Lin's Concordance Correlation Coefficient.*

Description

Lin's Concordance Correlation Coefficient.

Usage

```
ccc(obs, pred)
```

Arguments

`obs`, `pred`	Numeric vectors of the same length.

Value

A single numeric, or NA when fewer than two finite pairs remain.

Examples

```
obs <- c(10, 20, 30, 40)
ccc(obs, obs)        # 1: perfect agreement
ccc(obs, obs + 5)    # below 1: an offset is disagreement too, though r is 1
```

check_fold_plan — *Check a plan's folds are well formed, and describe them.*

Description

Run this before any training: a broken plan caught here costs a second, caught later it costs the whole run. Checks that, within each fold, train and validation are disjoint; that test (when present) appears in neither; and that across the folds every pooled row is validated exactly once – the property that makes the k metrics a partition of the pool rather than an arbitrary set of overlapping subsets.

Usage

```
check_fold_plan(plan, meta = NULL, group = "auto")
```

Arguments

plan	A fold_plan.
meta	The point table the plan was made for. Given, the grouping is proven against it: no group may be split across train and validation.
group	How the rows group, as in holdout().

Value

tibble, one row per fold.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
plan <- spatial_folds(meta, k = 3, test_frac = 0.2, block_size = 0.05)
check_fold_plan(plan, meta)
```

compare_importance — *Several importances, side by side.*

Description

Aligns importances computed by different methods, variants, rows or runs – permutation over all rows against within blocks, the test set against the folds, the spatial design against the random one – by what each perturbed (the variable, for a permutation), and says how far their rankings agree. Where they disagree is the reading: a variable that keeps its importance within blocks discriminates locally; one that loses it carries a regional gradient.

Usage

```
compare_importance(..., n = 20L)
```

Arguments

`...`	Two or more `dsm_importance`, named or not. Names become the labels; unnamed ones are labelled by run, rows and method.
`n`	How many variables the print shows.

Value

An `importance_comparison`: `table` (one row per target: each importance, its rank, and its share of that importance's largest), `agreement` (Spearman's correlation between the rankings, over the variables they share), `labels`, and for two SHAP importances of the same points `points_agreement` (per variable, the correlation of their values point by point and their mean difference against the second's size).

Examples

```
run <- example_run()    # a small fitted run, made once a session
perm <- dsm_importance(run$final, run$data, permutation_importance(draws = 2),
                       verbose = FALSE)
shap <- dsm_importance(run$final, run$data, shap_importance(samples = 20, background = 20),
                       verbose = FALSE)
compare_importance(permutation = perm, shap = shap)
```

`conformal_calibrate` *Calibrate a conformal interval from held-out residuals.*

Description

Calibrate a conformal interval from held-out residuals.

Usage

```
conformal_calibrate(obs, pred, alpha = 0.1, difficulty = NULL, group = NULL)
```

Arguments

`obs, pred`	Observations and predictions on the CALIBRATION set – data the model did not train on. Using training residuals produces intervals that are too narrow by exactly the amount the model overfits, and the result looks like a much better map.
`alpha`	Miscoverage rate: 0.1 asks for 90% coverage.

`difficulty`	Optional per-point difficulty score (e.g. the ensemble spread). When given, intervals scale with it. Must be positive.
`group`	Optional group of each point – a spatial block, a region, a profile. Given, every group weighs the same in the quantile, whatever its number of points: q is the smallest score at which the mean of the groups' empirical distributions reaches (1 - alpha)(1 + 1/m), m groups (Dunn, Wasserman & Ramdas 2023). The interval then speaks for a new point of a new group rather than for a point drawn from the pooled sample.

Value

An object of class "conformal_cal".

Examples

```
set.seed(1)
obs  <- rlnorm(300, 3, 0.4)
pred <- obs * exp(rnorm(300, 0, 0.25))
cal  <- conformal_calibrate(obs[1:150], pred[1:150], alpha = 0.1)
cal
# the same residuals, every one of 30 groups weighing alike
conformal_calibrate(obs[1:150], pred[1:150], alpha = 0.1, group = rep(1:30, 5))
```

`conformal_cv` *Calibrate on the folds of a resampling run, then report coverage.*

Description

The natural calibration set in this framework is the VALIDATION side of each fold: it is held out of training, it is separated spatially with a buffer, and it already exists. This walks the per-unit predictions a run wrote, calibrates on one fold and measures coverage on the others, so the reported PICP is always out-of-calibration.

Usage

```
conformal_cv(pred_obs, alpha = 0.1, difficulty = NULL, group = NULL)
```

Arguments

`pred_obs`	A table with columns fold, obs, pred (as the runners write).
`alpha`	Miscoverage rate.
`difficulty`	Optional column name holding a difficulty score.
`group`	Optional column name to break coverage down by.

Value

A picp_report over the pooled out-of-calibration points.

Examples

```
set.seed(1)
pred_obs <- data.frame(fold = rep(1:5, each = 60), obs = rlnorm(300, 3, 0.4))
pred_obs$pred <- pred_obs$obs * exp(rnorm(300, 0, 0.25))
conformal_cv(pred_obs, alpha = 0.1)
```

---

conformal_interval *Turn predictions into intervals.*

Description

Turn predictions into intervals.

Usage

```
conformal_interval(cal, pred, difficulty = NULL, lower_limit = -Inf)
```

Arguments

`cal`	A conformal_cal.
`pred`	Predictions to wrap.
`difficulty`	Difficulty scores for these predictions. Required when the calibration was normalised, and refused when it was not – mixing the two silently produces intervals with no guarantee at all.
`lower_limit`	Floor for the lower bound, e.g. 0 for a stock. Clipping a bound at a physical limit can only INCREASE coverage, so the guarantee survives it.

Value

A tibble: pred, lower, upper and width, one row per prediction.

Examples

```
set.seed(1)
obs  <- rlnorm(300, 3, 0.4)
pred <- obs * exp(rnorm(300, 0, 0.25))
cal  <- conformal_calibrate(obs[1:150], pred[1:150], alpha = 0.1)
iv <- conformal_interval(cal, pred[151:300], lower_limit = 0)
head(iv)
```

conformal_scale_fit — *Fit the scale of an interval: |residual| on covariates that say how wrong.*

Description

The scale of `conformal_scaled_calibrate()`, fitted on its own: a + b1 x1 + b2 x2 + ..., by least squares on |obs - pred|, with a floor. Fitted on points the quantile is NOT taken on – a calibration set's quantile, scaled by a fit to the cross-validated residuals, keeps the whole calibration set for q; fitted and calibrated on the same points, the fit would absorb the residuals the quantile measures.

Usage

```
conformal_scale_fit(obs, pred, covariates, floor_frac = 0.05)
```

Arguments

`obs, pred`	Observations and predictions, NATIVE units.
`covariates`	A data frame of scale covariates, one row per point, e.g. data.frame(level = pred, di = di).
`floor_frac`	The scale is never below this share of the median |residual|. A linear fit can go to zero or below at the edge of the covariates' range, and a zero-width interval there would claim a certainty the data never gave.

Value

A `conformal_scale`: the coefficients, the floor, and how well the fit explained |residual| (r2_fit).

Examples

```
set.seed(1)
level <- runif(300, 1, 4)
di    <- runif(300)                      # a dissimilarity index
obs   <- exp(level + rnorm(300, 0, 0.1 + 0.3 * di))
pred  <- exp(level)
conformal_scale_fit(obs, pred, data.frame(level = pred, di = di))
```

conformal_scaled_calibrate — *Calibrate an interval whose width is a fitted scale.*

Description

Calibrate an interval whose width is a fitted scale.

Usage

```
conformal_scaled_calibrate(
  obs,
  pred,
  covariates,
  alpha = 0.1,
  fit_frac = 0.5,
  floor_frac = 0.05,
  seed = 42L,
  scale = NULL,
  group = NULL
)
```

Arguments

obs, pred	Calibration observations and predictions, NATIVE units.
covariates	A data frame of scale covariates, one row per point, e.g. data.frame(level = pred, di = di). The scale is a + b1 x1 + b2 x2 + ..., fitted by least squares on |obs - pred|.
alpha	Miscoverage rate: 0.1 asks for 90%.
fit_frac	Share of the points the scale is fitted on; the rest calibrate q. One half each is the textbook split. Not used when `scale` is given.
floor_frac	The scale is never below this share of the median |residual| of the fitting half. A linear fit can go to zero or below at the edge of the covariates' range, and a zero-width interval there would claim a certainty the data never gave.
seed	Seed of the split.
scale	NULL: the scale is fitted on `fit_frac` of these points and q taken on the rest. A `conformal_scale` from `conformal_scale_fit()`, fitted on OTHER points: every point here then calibrates q.
group	Optional group of each point, as in `conformal_calibrate()`: q weighs every group alike, and the split, when there is one, keeps a group on one side.

Value

A `conformal_scaled` (also a `conformal_cal`).

Examples

```
set.seed(1)
level <- runif(300, 1, 4)
di    <- runif(300)                      # a dissimilarity index
obs   <- exp(level + rnorm(300, 0, 0.1 + 0.3 * di))
pred  <- exp(level)
covariates <- data.frame(level = pred, di = di)
cal <- conformal_scaled_calibrate(obs[1:200], pred[1:200], covariates[1:200, ],
                                  alpha = 0.1)
cal
```

```
# the scale fitted on other points, and all 100 of these calibrating q
sc <- conformal_scale_fit(obs[1:200], pred[1:200], covariates[1:200, ])
conformal_scaled_calibrate(obs[201:300], pred[201:300], covariates[201:300, ],
                           scale = sc)
```

`conformal_scaled_interval`
Turn predictions into intervals with a fitted scale.

Description

Turn predictions into intervals with a fitted scale.

Usage

```
conformal_scaled_interval(cal, pred, covariates, lower_limit = -Inf)
```

Arguments

`cal`	From conformal_scaled_calibrate().
`pred`	Predictions, native units.
`covariates`	The same covariates the scale was fitted on, for these predictions – the level, and the dissimilarity index computed the same way as the calibration points' was.
`lower_limit`	Floor of the lower bound (0 for a stock).

Value

A tibble: pred, lower, upper and width, one row per prediction.

Examples

```
set.seed(1)
level <- runif(300, 1, 4)
di    <- runif(300)                      # a dissimilarity index
obs   <- exp(level + rnorm(300, 0, 0.1 + 0.3 * di))
pred  <- exp(level)
covariates <- data.frame(level = pred, di = di)
cal <- conformal_scaled_calibrate(obs[1:200], pred[1:200], covariates[1:200, ],
                                  alpha = 0.1)
iv <- conformal_scaled_interval(cal, pred[201:300], covariates[201:300, ],
                                lower_limit = 0)
picp_report(obs[201:300], iv$lower, iv$upper)
```

context_importance *Context importance: how far from the point, and at which scale, the model reads.*

Description

The patch is a neighbourhood. This permutes part of it – the point's own pixel, a ring of pixels at one distance from it, every ring but the centre, or one window's whole input – and measures the loss of skill, as `permutation_importance()` does for a variable. The windows are cut around the same point, so a ring is the same ground in every window, and it takes its values from one donor in all of them.

Usage

```
context_importance(
  by = c("ring", "window"),
  bands = NULL,
  per_variable = FALSE,
  draws = 5L,
  metric = c("ccc", "rmse", "rmse_transform"),
  seed = 42L
)
```

Arguments

`by`	"ring" (the default): one row per distance band. "window": one row per window the model reads, that window's whole input permuted – how much the branch carries – and, when the model has a gate between its two branches, the gate read point by point beside it (`$gate`).
`bands`	For `by = "ring"`: the bands, a named list of ring numbers – 0 is the point's own pixel, ring d the 8d pixels d steps from it. NULL: the centre, each ring alone, and the context (every ring but the centre); with `per_variable`, the centre and the context.
`per_variable`	For `by = "ring"`: one row per variable and band – which variables the model reads through the neighbourhood, and which at the point alone. The variables are those of `groups` in `dsm_importance()`.
`draws`	Permutations per band (or variable and band, or window) and per model.
`metric`	What the table is ranked by, as in `permutation_importance()`.
`seed`	Seed of the permutations, shared by every model.

Details

A large importance of the context says the network reads the neighbourhood. It does not say the neighbourhood adds what the centre lacks: at 250 m most covariates are smooth, a pixel three cells away is close to a copy of the centre, and a network can lean on the rim and learn nothing the centre would not have told it. Ring d holds 8d pixels, so the table gives the cost per pixel beside the total: a wider ring costs more for its area alone.

Value

An `importance_spec`, for `dsm_importance()`.

Examples

```
context_importance(by = "ring")
```

---

cv_plus_calibrate *Calibrate a CV+ interval from the folds of a cross-validation.*

Description

Calibrate a CV+ interval from the folds of a cross-validation.

Usage

```
cv_plus_calibrate(obs, pred, fold, alpha = 0.1, difficulty = NULL)
```

Arguments

`obs`	Observations of the calibration points: every point validated once, in one fold.
`pred`	Each point's out-of-fold prediction – by the model of the fold that held it out (for a seed ensemble, the median of that fold's seeds).
`fold`	The fold that held each point out.
`alpha`	Miscoverage rate: 0.1 asks for 90%.
`difficulty`	Optional positive scale of each point, for a normalised CV+: the residuals are divided by it, and `cv_plus_interval()` multiplies them back by the new point's.

Value

A `cv_plus_cal`: each fold's sorted scores, the rank, the folds.

Examples

```
set.seed(1)
x <- runif(200)
fold <- rep(1:5, 40)
y <- 10 + 5 * x + rnorm(200)
# each fold's model: a line fitted without that fold
fits <- lapply(1:5, function(k) lm(y ~ x, data = data.frame(x, y)[fold != k, ]))
oof  <- vapply(seq_along(y), function(i)
  unname(predict(fits[[fold[i]]], data.frame(x = x[i]))), numeric(1))
cal <- cv_plus_calibrate(y, oof, fold, alpha = 0.1)
cal
# the five models at three new points, one column per fold
new <- data.frame(x = c(0.1, 0.5, 0.9))
cv_plus_interval(cal, sapply(fits, predict, newdata = new))
```

cv_plus_interval — *Turn the fold models' predictions into CV+ intervals.*

Description

Turn the fold models' predictions into CV+ intervals.

Usage

```
cv_plus_interval(cal, fold_pred, difficulty = NULL, lower_limit = -Inf)
```

Arguments

`cal`	A `cv_plus_cal`, from `cv_plus_calibrate()`.
`fold_pred`	A matrix, one row per new point and one column per fold: that fold's model at the point (for a seed ensemble, the median of the fold's seeds). Columns named by fold are matched by name; unnamed ones are read in the order of `cal$folds`.
`difficulty`	The new points' scale, for a normalised calibration; refused for a plain one, as in `conformal_interval()`.
`lower_limit`	Floor of the lower bound, e.g. 0 for a stock.

Value

A tibble: lower, upper and width, one row per new point.

Examples

```
set.seed(1)
fold <- rep(1:4, 25)
oof  <- rnorm(100, 20, 3)              # each point's out-of-fold prediction
obs  <- oof + rnorm(100, 0, 2)
cal  <- cv_plus_calibrate(obs, oof, fold, alpha = 0.1)
# the four fold models at two new points: they disagree a little
cv_plus_interval(cal, rbind(c(19.5, 20.2, 20.0, 20.4), c(30.1, 29.0, 29.8, 30.6)),
                 lower_limit = 0)
```

cv_residuals — *Cross-validated residuals from a tuning run, for calibration.*

Description

Every point appears once as validation, pooled over folds and averaged over seeds, so the residual distribution covers the whole study area instead of one fold's worth of it.

Usage

```
cv_residuals(run_dir, config_id, role = "validation")
```

Arguments

run_dir	Tuning run directory (the one holding predictions/).
config_id	Which config's predictions to read.
role	Which role to keep. "validation" is the point of this.

Value

A tibble with sample_id, obs, pred (the seed ensemble's median), and n_seeds; or NULL when the run wrote no usable predictions.

Examples

```
run <- example_run()    # a small fitted run, made once a session
res <- cv_residuals(run$fit$run_dir, run$final$selected_config_ids)
head(res)
conformal_calibrate(res$obs, res$pred, alpha = 0.1)
```

---

cv_residuals_for_config

Cross-validated residuals of a configuration, from any tuning run.

Description

Cross-validated residuals of a configuration, from any tuning run.

Usage

```
cv_residuals_for_config(run_dir, cfg_row, required = TRUE)
```

Arguments

run_dir	A tuning run directory (tune_grid.rds, predictions/).
cfg_row	One row of a grid: the configuration, whatever it is called in that run.
required	TRUE stops, saying why, when the run cannot serve the configuration; FALSE says why and returns NULL.

Value

What cv_residuals() returns, with the run's own config_id attached as attribute "config_id".

Examples

```
run <- example_run()    # a small fitted run, made once a session
grid <- readRDS(file.path(run$fit$run_dir, "tune_grid.rds"))
cfg  <- grid[grid$config_id == run$final$selected_config_ids, ]
# found by its hyperparameters, so the same call reads any other tuning run
head(cv_residuals_for_config(run$fit$run_dir, cfg))
```

---

di_reference *Summarise a training set for dissimilarity-index computation.*

Description

Computed once and reused for every prediction chunk: the training data does not change, and the pairwise mean is the expensive part.

Usage

```
di_reference(x_train, weights = NULL, max_pairs = 2e+06, seed = 42L)
```

Arguments

x_train	Numeric matrix of SCALED training predictors (n x p), in the model's channel order.
weights	Optional per-predictor weights, e.g. variable importance. NULL gives every predictor the same weight, which is what Meyer & Pebesma fall back to when no importance is available.
max_pairs	Cap on the number of point pairs used to estimate avg_dist. The full n^2 is unnecessary for a mean and quadratic in memory; a large random sample estimates it to more precision than the quantity deserves.
seed	Draw seed for that sample.

Value

An object of class "di_reference".

Examples

```
set.seed(1)
x_train <- matrix(rnorm(200), 100, 2)    # 100 training rows of 2 scaled predictors
ref <- di_reference(x_train)
ref
```

dissimilarity_index *The dissimilarity index of new data.*

Description

The dissimilarity index of new data.

Usage

```
dissimilarity_index(ref, x, chunk = 2000L)
```

Arguments

ref	From di_reference().
x	Numeric matrix of new rows, scaled with the MODEL's scaling.
chunk	Rows of x compared at a time. It bounds the memory a call takes and does not change the result.

Value

Numeric vector, one DI per row. 0 means identical to a training point; 1 means as far from the training data as two training points are from each other on average.

Examples

```
set.seed(1)
x_train <- matrix(rnorm(200), 100, 2)    # 100 training rows of 2 scaled predictors
ref <- di_reference(x_train)
dissimilarity_index(ref, rbind(c(0, 0), c(4, 4)))   # near the data, and far from it
```

dsm_final *Refit the configuration a tuning run selects, under N seeds.*

Description

Refit the configuration a tuning run selects, under N seeds.

Usage

```
dsm_final(
  tuning,
  data = NULL,
  config = "auto",
  rule = c("one_se", "rank1"),
  metric = "val_ccc",
```

```
  seeds = 10L,
  validation_frac = 0.15,
  predpoints = NULL,
  training = list(),
  transform = NULL,
  n_cores = NULL,
  threads_per_unit = 5L,
  max_ram_gb = NULL,
  conformal_alpha = c(0.1, 0.05),
  intervals = c("cv", "split", "cv_plus"),
  strata = NULL,
  output_dir = NULL,
  run_id = NULL,
  resume = TRUE,
  verbose = TRUE
)
```

Arguments

`tuning` A `dsm_fit` from dsm_train(), or the directory of a tuning run (it holds fold_plan.rds, tune_grid.rds and comparison/).

`data` From dsm_load(): the store the tuning ran on. Taken from the dsm_fit when `tuning` is one. Any windows may be loaded – the workers read the ones the selected configuration needs.

`config` "auto" applies `rule`; config id(s) choose by hand, and the frozen record then says "manual".

`rule` "one_se" (the simplest config within one standard error of the best) or "rank1" (the best mean).

`metric` The selection metric, as in the tuning table.

`seeds` A count (seeds 42, 43, ...) or the seeds themselves.

`validation_frac` Share of the non-test rows the refit stops on, carved by the tuning plan's own criterion (refit_split()): blocks, random rows, whole regions, or kNNDM against the same prediction points.

`predpoints` For a kNNDM tuning run made before its plan kept its prediction points (2026-09-28): those points, a data frame with x and y. NULL takes the plan's.

`training` Overrides of the refit schedule (.final_training_defaults) – any argument of train_one_cnn().

`transform` NULL for the store's own inverse (see dsm_train()).

`n_cores` Cores for the whole fit. NULL is the physical cores minus one.

`threads_per_unit` Threads each seed trains with. Part of the result (see the header), so it is recorded; 5 was measured best here (T1, T2).

`max_ram_gb` RAM the workers may use in total. NULL for 70% of what is available at the start (read with the ps package).

`conformal_alpha` Miscoverage levels of the intervals: 0.1 is 90%.

`intervals` Which interval calibrations to make and check on the test set (each at a constant width and at a width fitted on the level and the dissimilarity index; "cv" and "split" also with every group weighing alike):

- "cv": on the tuning run's cross-validated residuals – no data spent, no guarantee, the configuration having been chosen on those folds;
- "split": split conformal, on the calibration set the tuning plan held out (`calibration_frac` in `spatial_cv()` and the others) – coverage at least 1 - alpha for points exchangeable with it (Lei et al. 2018);
- "cv_plus": CV+ (Barber et al. 2021), the tuning run's fold models at each point – at least 1 - 2 alpha - sqrt(2/n) for an algorithm fixed in advance, which a configuration chosen on the folds is only nearly. A method the runs cannot support is left out and said so.

`strata` NULL, or a named list of labels, one per point of the store (e.g. `list(ecoregion = eco)`), to read the test coverage within, beside the area of applicability and the fifths of the predicted level.

`output_dir` Where runs go. NULL for final_model/ beside the tuning run's directory.

`run_id` NULL for `final_<timestamp>`. Give an existing one with resume = TRUE to finish an interrupted fit.

`resume` Skip seeds whose checkpoint and record are already there. A resumed run is held to the settings it started with (run_spec.rds): another thread count, schedule, grid, split or scaling is refused, and so is a directory dsm_final() did not start.

`verbose` Report progress, and print the result.

Value

A `dsm_final`, printed with the report.

Examples

```
run <- example_run()    # a small fitted run, made once a session
final <- dsm_final(run$fit, seeds = 2, n_cores = 1, threads_per_unit = 1,
                   training = list(n_epochs = 10, patience = 5), output_dir = tempdir(),
                   run_id = "final_example", verbose = FALSE)
final
```

`dsm_importance` *Variable importance of a fitted model, by the method you choose.*

Description

The method is an argument, as the validation design is in dsm_train(): permutation_importance() measures how much the model relies on each variable, over all rows or within blocks or classes; context_importance() how far from the point, and through which window, it reads; shap_importance() how each variable pushes each prediction, up or down; sage_importance() how much of the model's skill each variable carries, shared fairly among variables that carry the same information; refit_importance() how much skill a network trained without the variable loses; ale_effect() how the prediction changes along each variable's range. Each variable is a channel, or the channels of one categorical (see importance_groups()), or a group of yours.

Usage

```
dsm_importance(
  final,
  data,
  method = permutation_importance(),
  rows = c("test", "folds"),
  groups = "auto",
  config = NULL,
  threads = NULL,
  batch_size = 512L,
  verbose = TRUE,
  at = NULL,
  rasters = NULL,
  seeds = NULL,
  chunk_points = 5000L
)
```

Arguments

final	A dsm_final, or the directory of a final run.
data	The dsm_data the run was fitted on, from dsm_load(). Its windows need not be loaded: one that is not is read from the store.
method	An importance method: permutation_importance(), context_importance(), shap_importance(), sage_importance(), refit_importance() or ale_effect().
rows	"test" (the default): the final run's seeds, on the test set the whole run held out – the importance of the model that draws the map. "folds": the tuning run's models of the same configuration, each on its own fold's validation rows – the importance as that validation design sees it; run it for each design to compare them.
groups	Which channels form one variable: "auto", "channel", or your grouping, as in importance_groups().
config	Which of the final run's configurations, when it fitted more than one. NULL: the selected one.
threads	torch threads for this session; NULL leaves them as set.
batch_size	Points per forward pass.

verbose	Say what is done, model by model.
at	Points of the map instead of rows of the store: a data frame of x, y, from importance_points() or of your own. Their patches are cut from the rasters as the profiles' were, and only shap_importance() runs there – a map point has no observation to score against. The values become maps with importance_map().
rasters	With at: where the rasters are, as in dsm_predict(); NULL for the store's own raster table.
seeds	Which of the final run's seeds to use; NULL for all. A map of many points may take a few: each seed is one model explained.
chunk_points	With at: points cut and explained at a time. Their patches are held in memory together.

Details

Before anything is perturbed, each model's own score is computed again and must reproduce the one its run wrote; if it does not, nothing is measured.

Value

A dsm_importance: table (one row per target – a variable, a band, a variable at a band or a window – ranked), by_model (one row per target and model), baseline (each model's unperturbed score), groups, targets (what each target is), gate (by window, for a model with a gate: the gate point by point), units (the models scored) and the settings. For SHAP: table (mean |SHAP|, its share and direction per variable), points (every point's values), patch and pixels (where in the patch they sit) and completeness (how well they add up). For ALE: table (the spread, trend and extremes of each effect), curves (each continuous variable's curve, with its spread between models) and classes (each categorical's class effects). For SAGE: table (the loss each variable takes away, and its share), by_model and losses (each model's loss, and the mean prediction's, which SAGE splits). For a refit: table, by_model, raw (each unit's scores), check (the seeds trained again with nothing left out, against the run), noise (what a refit moves a score by with nothing to lose) and refit_dir.

Examples

```
run <- example_run()    # a small fitted run, made once a session
imp <- dsm_importance(run$final, run$data, permutation_importance(draws = 2),
                      verbose = FALSE)
imp
plot(imp)
```

`dsm_load` *Load everything a run needs, and check that it fits together.*

Description

Replaces the six-step preamble every pipeline script used to repeat: open the store, read the points, read the predictor types, align the points to the store, read the raster resolution, and verify the store can serve this configuration.

Usage

```
dsm_load(
  patch_dir,
  points = NULL,
  type_table = NULL,
  windows = NULL,
  cell_size = NULL,
  raster_table = NULL,
  target_col = NULL,
  verbose = TRUE
)
```

Arguments

`patch_dir`	Directory written by the extraction step – or the `dsm_store` dsm_prepare() returned, in which case nothing else is needed.
`points`	Point table, or a path to the CSV written by stage 01. NULL for a store written by dsm_prepare(), which carries its own copy.
`type_table`	Predictor types, or a path to its CSV. NULL as above.
`windows`	Windows to load. NULL loads every window the store has, which is the wrong default when the grid needs two of five – pass the windows the grid actually uses and the store reads only those. integer(0) loads none: the store's table alone, from which a fold cache reads each window it needs from its file (dsm_final()'s workers do).
`cell_size`	Raster resolution in x/y units. NULL reads it from `raster_table`, which is the only source that cannot drift.
`raster_table`	Path to raster_table_used.csv, for `cell_size`.
`target_col`	Expected target column, for the store lock.
`verbose`	Print the loaded data.

Value

A `dsm_data` object. `$transform` is the target transform the store was built under – name, forward and inverse – or NULL when the store did not record one.

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
                     percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                     overwrite = TRUE, verbose = FALSE)
dsm_load(store, windows = integer(0))   # the table alone: loading windows needs torch
```

`dsm_predict` *Predict a map from a final model.*

Description

Predict a map from a final model.

Usage

```
dsm_predict(
  final,
  data,
  rasters = NULL,
  qc_table = NULL,
  extent = NULL,
  config = NULL,
  calibration = NULL,
  aoa_weights = NULL,
  alpha = 0.1,
  intervals = NULL,
  weighting = c("point", "group"),
  clamp = NULL,
  bands = NULL,
  engine = c("auto", "patch"),
  output_dir = NULL,
  run_id = NULL,
  resume = TRUE,
  n_cores = NULL,
  threads_per_worker = 5L,
  max_ram_gb = NULL,
  unit_rows = NULL,
  step_rows = NULL,
  chunk_cols = 2048L,
  probe = TRUE,
  verbose = TRUE
)
```

Arguments

`final`	A `dsm_final`, or the directory of a final run (dsm_final() or stage 04).
`data`	From dsm_load(), or the directory of a store written by dsm_prepare() – only its tables are read, no patches.
`rasters`	NULL for the rasters the store was extracted from; a directory holding the same file names (a coarser grid, for a cheap pass end to end); or a table (data frame or CSV path) with columns predictor and raster_file.
`qc_table`	NULL for the store's own QC rules; a data frame or CSV path for a store written before dsm_prepare().
`extent`	NULL for the whole grid; a SpatExtent, c(xmin, xmax, ymin, ymax), anything terra::ext() reads, or list(rows = c(a, b), cols = c(c, d)). Pixels at the extent's edge are predicted with their real neighbours, so a part of the map is the same numbers as that part of the whole.
`config`	NULL for the final run's selected configuration.
`calibration`	Where the intervals, the smearing factor and the AOA come from: a named vector of tuning-run directories whose cross-validated residuals calibrate them, e.g. `c(block = "<spatial CV run>", knndm = "<kNNDM run>")`. Each source gets its own bands. NULL for the final run's own tuning run; character(0) for none (then no interval, DI or AOA).
`aoa_weights`	NULL: every channel weighs alike in the dissimilarity index. Or a `dsm_importance` (see `importance_weights()`), or one non-negative weight per channel, named or in the model's order: the DI, the AOA and the level+DI interval then measure a pixel's distance from the training data in what the model uses (Meyer & Pebesma 2021). A map resumed with other weights is refused.
`alpha`	Miscoverage of the intervals: 0.1 is 90%.
`intervals`	Which intervals the map carries (see `dsm_final()`, which checks all of them on the test set): "cv", calibrated on each source's cross-validated residuals; "split", split conformal on the final run's calibration set; "cv_plus", CV+ with each source's fold models – which predicts every pixel once more per fold model, the map's dearest band by far. NULL: "cv", and "split" when the final run has a calibration set.
`weighting`	"point" (every calibration point weighs the same) or "group" (every group of the plan does – block, region, profile; see `conformal_calibrate()`), for the "cv" and "split" intervals. CV+ is by point.
`clamp`	c(lower, upper) of a prediction in native units. NULL for the refit's own (its evaluation clamps to c(0, Inf) by default).
`bands`	NULL for all, or some of: ensemble_median, ensemble_mean, ensemble_sd, ensemble_mad, ensemble_min, ensemble_max, smeared_mean, interval_constant, interval_level_di, di, aoa, valid_mask.
`engine`	"auto": fully convolutional where that gives exactly the patch-by-patch numbers for the network, patch by patch elsewhere. "patch": patch by patch everywhere – slow, for checks.
`output_dir`	Where maps go. NULL for `<final run>/maps`.

run_id — NULL for map_<timestamp>. An existing one resumes.

resume — Keep the units already finished.

n_cores — Cores for the whole map. NULL for the physical cores minus one.

threads_per_worker — Torch threads each worker runs with.

max_ram_gb — RAM all workers may use together. NULL for 70% of what is free when the map starts (read with the ps package).

unit_rows, step_rows, chunk_cols — The geometry of the work: a unit is unit_rows rows over the whole extent width, read step_rows rows at a time (a multiple of 16), with the network run over chunk_cols columns at a time. NULL sizes the first two from the RAM.

probe — Predict the profiles' own pixels first and compare with the final run's stored predictions (see above). A failure stops the map.

verbose — Report progress, and print the result.

Value

A dsm_prediction, printed.

Examples

```
run <- example_run()    # a small fitted run, made once a session
map <- dsm_predict(run$final, run$data, output_dir = tempdir(), run_id = "map_example",
                   n_cores = 1, threads_per_worker = 1, verbose = FALSE)
map
```

---

dsm_prepare — *Build a patch store from a point table and a folder of aligned rasters.*

Description

Build a patch store from a point table and a folder of aligned rasters.

Usage

```
dsm_prepare(
  points,
  target,
  raster_dir,
  windows,
  out_dir = NULL,
  store_dir = NULL,
  metadata_dir = NULL,
```

```
  points_file = NULL,
  profile_id = "profile_id",
  coords = c("x", "y"),
  crs = NULL,
  percentage = NULL,
  dummy = "auto",
  drop = NULL,
  na_below = NULL,
  percentage_limits = c(0, 100),
  transform = c("none", "log1p"),
  target_min = NULL,
  subsample = NULL,
  target_label = NULL,
  target_unit = NULL,
  raster_pattern = "\\.tif$",
  chunk_nrows = 1000L,
  n_cores = NULL,
  max_ram_gb = NULL,
  read_gap = 256L,
  read_max_cols = 4096L,
  overwrite = FALSE,
  verbose = TRUE
)
```

Arguments

`points` An sf object, a data.frame with coordinate columns, or a path to a spatial file sf can read (GPKG, shapefile, GeoJSON).

`target` Name of the target column in `points`.

`raster_dir` Folder of aligned single-band rasters, one predictor per file. Only files at the top level matching `raster_pattern` are read.

`windows` Patch sizes in pixels, odd. REQUIRED: a window's ground extent is window x resolution, so there is no default that is right at every resolution. 3/9/15 were chosen for 250 m.

`out_dir` Where to write, as out_dir/patches (the store), out_dir/metadata (tables for a human) and out_dir/points.csv. Or give the three separately with store_dir, metadata_dir and points_file.

`store_dir`, `metadata_dir`, `points_file` Where each part goes when it is not under `out_dir`: the store, the tables for a human, the point table.

`profile_id` Column that identifies an observation. Rows sharing it are de-duplicated (the first is kept) and later kept in the same fold. Absent, every row is its own profile.

`coords`, `crs` For a data.frame: the coordinate columns and their CRS. NULL crs means the coordinates are already in the rasters' CRS.

`percentage` Regular expressions over the CLEANED raster names for channels in 0-100. Anchor them (^pnv_, ^peatland_extent$) to avoid matching more than intended; every match is reported.

`dummy` "auto" to detect 0/1 channels, or the channel names.

`drop` Channel names (raw or cleaned) to leave out.

`na_below` Named numeric: names are regexes over cleaned names, values are thresholds. A value <= its threshold becomes NA (a nodata sentinel such as -9999, or stage 01's temperature floor of -100).

`percentage_limits` Percentages are clamped into this range; NULL to leave them as read. The clamp keeps the slight overshoot of an interpolated surface instead of turning it into NA.

`transform` "none" or "log1p" – the space the model trains in.

`target_min` Rows with target <= target_min are dropped. NULL keeps any finite target the transform accepts.

`subsample` NULL, or list(frac, block_size, seed, strata): keep a fraction of the points in whole spatial blocks, for a run that finishes sooner. `strata`, optional, is the side of square strata, a whole multiple of `block_size`: each keeps its points up to one common quota, so the cut falls on the densest regions and no region is lost. Recorded, so a subsampled result is never mistaken for a full one.

`target_label` Name of the target in reports. NULL uses `target`.

`target_unit` Unit of the target, for reports. NULL leaves it unrecorded.

`raster_pattern` Regular expression the raster file names must match.

`chunk_nrows` Raster rows read per chunk. A performance knob: the store does not depend on it.

`n_cores` Cores for the extraction, one band per core. NULL uses the physical cores minus one, fewer if `max_ram_gb` cannot hold them. The result does not depend on it.

`max_ram_gb` RAM the extraction may use in total. NULL for 70% of what is available when it starts (read with the ps package; without it, no cap). Decides how many of the `n_cores` workers actually run.

`read_gap`, `read_max_cols` How the points of a row chunk are grouped into reads: windows closer than `read_gap` columns share a read, and no read is wider than `read_max_cols`. Performance knobs only – the store and the point table are identical whatever they are, and the test suite proves it.

`overwrite` A store_dir that already holds a store is refused unless TRUE, in which case that store's files are removed first.

`verbose` Report progress.

Value

A `dsm_store`, which dsm_load() accepts directly.

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
```

```
                      percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                      overwrite = TRUE, verbose = FALSE)
store
```

`dsm_report_final` *The declaration of a final model that already exists.*

Description

dsm_final() writes final_report.md as it fits. A final model fitted by stage 04, before dsm_final() existed, has the seeds and the files but no declaration – and retraining ten seeds to obtain one would also change them (T1: the thread count changes the numbers). This writes it from what is on disk, re-assembling the seeds the way P3 proved dsm_final() does, and changes nothing that is there: it ADDS final_report.md and selected_hyperparameters.csv to the run directory.

Usage

```
dsm_report_final(
  run_dir,
  tuning_dir,
  conformal_alpha = c(0.1, 0.05),
  verbose = TRUE
)
```

Arguments

`run_dir`	The final-model run (it holds comparison/final_run_summary.rds).
`tuning_dir`	The tuning run it was selected from.
`conformal_alpha`	As the run was calibrated; stage 04 used c(0.1, 0.05).
`verbose`	Print the result.

Value

A `dsm_final` describing the run, printed with the declaration.

Examples

```
run <- example_run()    # a small fitted run, made once a session
# the declaration of a final run, written from what is on disk
dsm_report_final(run$final$run_dir, run$fit$run_dir)
```

dsm_train *Fit and tune a model over a resampling plan.*

Description

Fit and tune a model over a resampling plan.

Usage

```
dsm_train(
  data,
  model = "cnn",
  resampling = spatial_cv(),
  tune_grid = NULL,
  tune_length = 20L,
  n_seeds = 3L,
  transform = NULL,
  clamp = c(0, Inf),
  features = c("centre", "window_mean"),
  output_dir,
  run_id = format(Sys.time(), "%Y%m%d_%H%M%S"),
  base_seed = 42L,
  device = NULL,
  n_cores = NULL,
  in_session = !is.null(device),
  threads_per_unit = 5L,
  max_ram_gb = NULL,
  test_ids = NULL,
  calibration_ids = NULL,
  resume = TRUE,
  evaluate_test = FALSE,
  verbose = TRUE,
  ...
)
```

Arguments

data	From dsm_load().
model	A registered model name ("cnn", "rf", "mlp", or anything registered with register_model()), or a model_spec.
resampling	A resample_spec, or a fold_plan to use as given.
tune_grid	An explicit grid. NULL asks the model for one – for the CNN, drawn over the windows the loaded store holds: every window alone and every pair.
tune_length	How many configurations to try when tune_grid is NULL. The same meaning as caret's: a budget, not a lattice.

`n_seeds` Repetitions per (config, fold). One gives a ranking with no error bar, which is a ranking of luck as often as of skill.

`transform` NULL (the default) uses the inverse of the transform the store was built under, which dsm_load() read (`data$transform`). A function is the inverse to use instead, and is refused if it disagrees with the store's.

`clamp` c(lower, upper), the plausible range of the target in native units: every prediction is clipped into it before it is scored. c(0, Inf) suits a stock or a concentration. A target that can be negative – a temperature, a log-ratio – needs c(-Inf, Inf): clipped at zero it loses half its predictions, and the metrics still look plausible. Refused if the data itself falls outside it. The run keeps it, and dsm_final() refits with the same.

`features` For tabular models: "centre", "window_mean", or both.

`output_dir` Where runs go; each run is a directory under it. Required: nothing is written where nobody said – a folder of your project, or `tempdir()` for a try.

`run_id` The run's directory name. Reusing one resumes that run (see `resume`); the default, a timestamp, always starts a fresh one.

`base_seed` Repetition s of every configuration trains under the seed base_seed + s - 1, the same for every configuration, so that two configurations differ by their hyperparameters and not by their luck.

`device` A torch device, to train the CNN on it in this session (see `in_session`). NULL builds one with setup_torch_device() when the units train in this session.

`n_cores` Cores for training. Side by side, the threads of all the workers together: n_cores / threads_per_unit workers, rounded down, and fewer if the RAM holds fewer. In this session, torch's threads for the CNN; the MLP's threads and ranger's for the forest too. NULL is the physical cores minus one (see resolve_cores()) – except that a `device` passed in keeps the threads it was set up with unless n_cores is given too.

`in_session` FALSE – the default, unless a `device` is given – trains the CNN's units side by side, each in an R process of its own with `threads_per_unit` threads. T2 measured three units of 5 threads 1.52x faster than one of 15, each giving exactly the numbers it gave alone. TRUE trains them one at a time in this session, on `device`, with `n_cores` threads; a CUDA device trains that way. The table models always train in this session.

`threads_per_unit` Threads each unit trains with side by side. It is part of a unit's result (T1), so the run records it and a resume with another count is refused. 5 was measured best here (T1, T2).

`max_ram_gb` RAM the side-by-side workers may use together. NULL for 70% of what is available when they start.

`test_ids` Sample ids forced into the test set.

`calibration_ids` Sample ids forced into the calibration set (see `calibration_frac` in `spatial_cv()`): in no fold, kept for the "split" intervals `dsm_final()` calibrates.

`resume` Keep the units of `run_id` that already finished, matched by their hyperparameters rather than their label. A fold plan or a clamp other than the run's is refused.

evaluate_test	Score the test set on every unit? FALSE, and deliberately so: a frozen test set stops being frozen once its score sits in the tuning table. It is scored once, after the choice, by freeze_selection() and score_test_grid().
verbose	Report progress.
...	Passed to the underlying runner (n_epochs, patience, ...).

Value

The runner's result, plus the plan and the data it used.

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
                     percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                     overwrite = TRUE, verbose = FALSE)
data <- dsm_load(store, verbose = FALSE)
fit <- dsm_train(data, resampling = spatial_cv(k = 2, block_size = 0.05, test_frac = 0.2),
                 tune_length = 2, n_seeds = 1, output_dir = tempdir(),
                 in_session = TRUE, n_cores = 2, n_epochs = 5, verbose = FALSE)
fit
```

---

equal_area_blocks *Square blocks of equal area, in kilometres.*

Description

Each point falls in a square cell of `size_km` on a Lambert azimuthal equal-area projection centred on the points (on the sphere of WGS84's authalic radius), so every block covers the same ground wherever it lies – which a grid in degrees does not: a degree of longitude is 111 km at the equator and 71 km at 50 degrees.

Usage

```
equal_area_blocks(x, y, size_km, coords = c("lonlat", "metres"), centre = NULL)
```

Arguments

x, y	Longitude and latitude in degrees (`coords = "lonlat"`), or easting and northing in metres of an equal-area projection of yours (`coords = "metres"`).
size_km	The side of a block, in km.
coords	"lonlat" (the default) or "metres".
centre	For "lonlat": the projection's centre, `c(longitude, latitude)`. NULL: the points' own means.

Value

A character vector, one block per point (the cell's column and row, "i_j"), with the size and the projection as attributes.

Examples

```
ex <- example_landscape()
blk <- equal_area_blocks(ex$profiles$x, ex$profiles$y, size_km = 2)
length(unique(blk))    # blocks holding a profile
table(table(blk))      # how many blocks hold 1, 2, ... profiles
```

---

example_landscape *A small synthetic landscape, for the examples and the tests.*

Description

80 x 80 cells of 0.0025 degrees (about 250 m) in coordinates of south-eastern Brazil – the place is borrowed, the landscape is made up – with eight predictors and 160 soil profiles, 90 of them in six survey clusters. The soil organic carbon stock (t/ha, 0-30 cm) was drawn from vegetation, clay, temperature, geology and the topographic position of the cell in its 7 x 7 neighbourhood: a network that reads the neighbourhood has something to find that the centre pixel does not show.

Usage

```
example_landscape()
```

Value

A list: `raster_dir`, the folder of the eight GeoTIFFs (elevation, temperature, precipitation, ndvi, clay_pct, and geology as three 0/1 dummies); `profiles`, a data frame of `profile_id`, `x`, `y`, `survey` and `soc_stock`; and `truth`, the formula the stock was drawn from.

Examples

```
ex <- example_landscape()
head(ex$profiles)
list.files(ex$raster_dir)
```

example_run — *A small fitted run on the example landscape, made once a session.*

Description

The store, a tuning run and a final model on `example_landscape()`, in `dir`: what the examples of the functions that need a fitted model start from. Made at the first call – a minute or two on one core – and kept for the session; later calls return it at once. It is an example, not a model: 160 profiles, one small configuration, 30 epochs.

Usage

```
example_run(dir = file.path(tempdir(), "soilcnn_example"), verbose = FALSE)
```

Arguments

`dir`	Where the run is made. The default is in `tempdir()`, so it goes with the session.
`verbose`	Report each step.

Value

A list: `data` (from `dsm_load()`), `fit` (from `dsm_train()`), `final` (from `dsm_final()`) and `dir`.

Examples

```
run <- example_run()
run$final
```

fold_leakage_report — *How much does each fold still leak?*

Description

Reports, per fold, the same quantities the pipeline check reports for the fixed split: the share of validation points sharing a raster cell with a training point, and sharing patch pixels at each window. Deliberately the SAME criterion, so a number here can be compared with a number there rather than being a second, incompatible notion of leakage.

Usage

```
fold_leakage_report(plan, meta, cell_size, windows = c(3L, 9L, 15L))
```

Arguments

plan	A fold_plan.
meta	Patch store meta with x/y.
cell_size	Raster resolution, in the units of x/y.
windows	Window sizes to report overlap for.

Value

A tibble with a `fold` column: the overlap shares per role and window, fold by fold.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
blocks   <- spatial_folds(meta, k = 3, block_size = 0.05)
buffered <- spatial_folds(meta, k = 3, block_size = 0.05, buffer = 7 * 0.0025)
fold_leakage_report(blocks, meta, cell_size = 0.0025, windows = c(3, 7))
fold_leakage_report(buffered, meta, cell_size = 0.0025, windows = c(3, 7))
```

---

freeze_selection *Record which config was chosen, and when.*

Description

Writes the selection into the run directory so a later test-set report can prove the choice preceded it. Call it at the moment the choice is made.

Usage

```
freeze_selection(
  run_dir,
  config_id,
  rule = "one_se",
  metric = "val_ccc",
  note = NA_character_
)
```

Arguments

run_dir	Tuning run directory (the one holding comparison/).
config_id	The chosen config (or configs, one per family).
rule	How it was chosen, e.g. "one_se" or "rank1".
metric	The selection metric.
note	Anything a reader would need to reconstruct the decision.

Value

The selection record, invisibly.

Examples

```
run_dir <- file.path(tempdir(), "tuning_example")
freeze_selection(run_dir, "cfg_002", note = "one_se on the cross-validation")
# the same choice again is accepted; another is refused
try(freeze_selection(run_dir, "cfg_005"))
```

---

get_model — *Fetch a registered model.*

Description

Fetch a registered model.

Usage

```
get_model(name)
```

Arguments

name	Registered name.

Value

The `model_spec` registered under `name`.

Examples

```
get_model("rf")
```

---

holdout — *A single train/validation/test split of a point table.*

Description

The fold constructor behind holdout_cv(). Whole groups are drawn, never single rows (see `group`), so the rows of one profile never straddle the split.

Usage

```
holdout(
  meta,
  validation_frac = 0.15,
  test_frac = 0.15,
  test_ids = NULL,
  seed = 42L,
  group = "auto",
  calibration_frac = 0,
  calibration_ids = NULL
)
```

Arguments

`meta` Point table (the patch store's meta): needs sample_id, and profile_id for `group = "auto"` to have anything to do.

`validation_frac` Share of the non-test pool that validates.

`test_frac` Share held out as the test set, drawn first.

`test_ids` Sample ids that ARE the test set, in place of `test_frac`: a test set drawn again on every run is not a test set.

`seed` Seed for this split only; the training seeds do not move it.

`group` "auto" keeps the rows of one profile_id together when the table repeats profiles; NULL or "row" makes each row its own unit; a column name, or a vector with one label per row, names the groups.

`calibration_frac` Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

`calibration_ids` Sample ids that ARE the calibration set, in place of `calibration_frac`, as `test_ids` is for the test set.

Value

A `fold_plan` with one fold.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
holdout(meta, validation_frac = 0.2, test_frac = 0.2)
# with a calibration set for the intervals beside the test set
holdout(meta, validation_frac = 0.2, test_frac = 0.2, calibration_frac = 0.15)
```

holdout_cv *A single train/validation/test split.*

Description

A single train/validation/test split.

Usage

```
holdout_cv(
  validation_frac = 0.15,
  test_frac = 0.15,
  group = "auto",
  seed = 42L,
  calibration_frac = 0
)
```

Arguments

`validation_frac`
: Share of the non-test pool that validates.

`test_frac`
: Share held out as the test set, drawn first.

`group`
: "auto" keeps the rows of one profile_id together when the table repeats profiles; NULL or "row" makes each row its own unit; a column name, or a vector with one label per row, names the groups.

`seed`
: Seed for this split only; the training seeds do not move it.

`calibration_frac`
: Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

Value

A `resample_spec`, as spatial_cv() returns.

Examples

```
holdout_cv(validation_frac = 0.2)
```

`importance_groups` *Which channels the importance treats as one variable.*

Description

A categorical raster arrives as several 0/1 channels, one per class. Permuting one of them alone makes points that belong to two classes or to none, so the channels of one categorical are one variable. The package is not told which raster a dummy came from, so it infers the sets: dummies that share a name prefix (cut at "_") and are never 1 together at the points. Every other channel is a variable of its own. The sets found are listed; check them, and give your own grouping when they are not what the rasters were.

Usage

```
importance_groups(data, groups = "auto")
```

Arguments

`data`	The `dsm_data` the model was fitted on, from `dsm_load()`.
`groups`	"auto" (the default), "channel" (every channel alone, dummies too – see above for what that measures), or your grouping: a data frame with columns `channel` and `variable`, or a named list (variable = channel names). Channels it does not name keep the automatic rule.

Details

A grouping of your own can be anything that makes sense to perturb as one: the classes of a categorical named outright, or themes (climate, relief, vegetation...), which answers how much the model relies on each theme as a block – the way to read many correlated variables.

Value

A tibble in channel order: `channel`, `variable`, and `rule` ("alone", "one-hot set" or "yours").

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
                     percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                     overwrite = TRUE, verbose = FALSE)
data <- dsm_load(store, windows = integer(0), verbose = FALSE)
importance_groups(data)          # the geology dummies, found as one variable
themes <- data.frame(channel = c("temperature", "precipitation"), variable = "climate")
importance_groups(data, themes)  # and two channels made one by you
```

`importance_map` *Maps of SHAP values, from an importance computed at points of the map.*

Description

Lays the values of `dsm_importance()(at = )` back on a grid: each cell the mean of the points in it. With the points' own grid (from `importance_points()`) every point is its own cell; with a coarser `resolution`, each cell averages the points it holds.

Usage

```
importance_map(x, resolution = NULL, output_dir = NULL)
```

Arguments

`x`	A `dsm_importance` from `shap_importance()` at map points.
`resolution`	Cell size of the maps, in the rasters' units; NULL for the points' own grid.
`output_dir`	Where to write the GeoTIFFs (shap.tif, one layer per variable; shap_dominant.tif and its legend, shap_dominant.csv; prediction.tif); NULL to only return them.

Value

An `importance_map`, a list: `shap` (one layer per variable: the mean SHAP value of the points in each cell, in the network's units), `dominant` (in each cell, the variable with the largest mean |SHAP|, as the number in `legend`), `legend`, `prediction` (the mean prediction, native units) and `files`. The three rasters are packed with `terra::wrap()`, so the map survives `base::saveRDS()`; `terra::unwrap()` gives each as a `SpatRaster`. `plot()` draws it.

Examples

```
run <- example_run()    # a small fitted run, made once a session
pts <- importance_points(run$final, run$data, every = 4)
imp <- dsm_importance(run$final, run$data, shap_importance(samples = 10, background = 20),
                      at = pts, seeds = 42, verbose = FALSE)
m <- importance_map(imp)
plot(m)
```

`importance_points` *Points of the map to explain: a regular grid of the model's raster cells.*

Description

The cells of the rasters the model was fitted on, every `every` rows and columns, inside `extent`, where the first channel has a value. With `every = 1` every cell – the map at full resolution, for a tile; with 40 at 250 m, one point every 10 km, for a region. The points carry their grid, so `importance_map()` can lay the values back on it.

Usage

```
importance_points(final, data, extent = NULL, every = 1L, rasters = NULL)
```

Arguments

`final`	A `dsm_final`, or the directory of a final run.
`data`	The `dsm_data` it was fitted on.
`extent`	c(xmin, xmax, ymin, ymax) in the rasters' coordinates; NULL for the whole raster.
`every`	Take every `every`-th cell, in rows and in columns.
`rasters`	Where the rasters are, as in `dsm_predict()`; NULL for the store's own raster table.

Value

A tibble of `x`, `y` (cell centres), with the grid as an attribute.

Examples

```
run <- example_run()    # a small fitted run, made once a session
pts <- importance_points(run$final, run$data, every = 4)   # every 4th cell
nrow(pts)
```

importance_weights — *One weight per channel, for the area of applicability, from an importance.*

Description

The dissimilarity index measures how far a pixel is from the training data, one axis per channel. Unweighted, a channel the model ignores counts as much as the one it leans on, and a pixel unlike the training data in an ignored channel falls outside the area of applicability for nothing. Meyer & Pebesma (2021) weight each axis by the predictor's importance; this gives each channel its variable's importance – every channel of a one-hot set, or of a group of yours, its variable's – and a negative importance, noise around zero, weight zero.

Usage

```
importance_weights(x)
```

Arguments

x	A `dsm_importance` with one value per variable: by `permutation_importance()`, `shap_importance()`, `sage_importance()`, `refit_importance()` or `ale_effect()`.

Value

A named numeric vector, one weight per channel in the store's order, for `dsm_predict(aoa_weights = )` or `aoa_reference(weights = )`.

Examples

```
run <- example_run()    # a small fitted run, made once a session
imp <- dsm_importance(run$final, run$data, shap_importance(samples = 20, background = 20),
                      verbose = FALSE)
importance_weights(imp)   # for dsm_predict(aoa_weights = ) or aoa_reference(weights = )
```

inside_aoa — *Is each row inside the area of applicability?*

Description

Is each row inside the area of applicability?

Usage

```
inside_aoa(di, threshold)
```

Arguments

di	From dissimilarity_index().
threshold	From aoa_threshold().

Value

Logical vector. TRUE means the cross-validated error applies here.

Examples

```
set.seed(1)
x_train <- matrix(rnorm(200), 100, 2)    # 100 training rows of 2 scaled predictors
ref <- di_reference(x_train)
threshold <- aoa_threshold(ref, folds = rep(1:5, 20))
inside_aoa(dissimilarity_index(ref, rbind(c(0, 0), c(4, 4))), threshold)
```

---

knndm_cv — *Folds matched to where the map will be predicted (kNNDM).*

Description

The better-founded alternative to spatial_cv(). Blocks need a block size and a buffer width, and nothing in the data says whether the chosen ones were right. kNNDM instead shapes the folds so the distance from a validation point to its nearest training point is distributed like the distance from a PREDICTION pixel to its nearest training point.

Usage

```
knndm_cv(
  k = 5L,
  predpoints = NULL,
  hold_out_test = FALSE,
  crs = 4326,
  project_to = "+proj=moll +lon_0=0 +datum=WGS84 +units=m",
  seed = 42L,
  hold_out_calibration = FALSE,
  ...
)
```

Arguments

k	Number of folds.
predpoints	Where the map will be predicted: a data frame with x and y, or an sf object. REQUIRED unless `modeldomain` is passed through `...` – the method has no meaning without it, so it is not defaulted.
hold_out_test	Carve a test set by running kNNDM with k + 1 folds and holding one out. FALSE by default: a test set that changes with k is not a frozen test set.

`crs`	CRS of `meta$x`/`meta$y`. 4326 by default.
`project_to`	Projection used for the distance comparison, or NULL to use the coordinates as they are (correct only if already projected).
`seed`	Seed, for the sampling kNNDM does internally.
`hold_out_calibration`	Carve a calibration set for the intervals as the test set is carved: kNNDM in k + 1 folds over what the test left, one held out. FALSE by default.
`...`	Passed to CAST::knndm() – `maxp`, `clustering`, `samplesize`, `modeldomain`, `space`.

Details

It therefore needs `predpoints`: a sample of where the map will be drawn. There is no default, because a default would turn the method into an expensive random split. prediction_sample(raster) produces one.

When the samples are well spread over the prediction area it converges by itself to ordinary random k-fold – it does not impose separation that prediction will not face. That is the property blocks cannot have.

Needs the CAST and sf packages. See R/knndm.R for the projection question, which is not optional on lon/lat data.

Value

A `resample_spec`, as spatial_cv() returns.

Examples

```
ex <- example_landscape()
elevation <- terra::rast(file.path(ex$raster_dir, "elevation.tif"))
knndm_cv(k = 5, predpoints = prediction_sample(elevation, size = 500))
```

`knndm_folds` *Fold plan from k-fold Nearest Neighbour Distance Matching.*

Description

Fold plan from k-fold Nearest Neighbour Distance Matching.

Usage

```
knndm_folds(
  meta,
  k = 5L,
  predpoints = NULL,
  test_ids = NULL,
  hold_out_test = FALSE,
```

```
  crs = 4326,
  project_to = "+proj=moll +lon_0=0 +datum=WGS84 +units=m",
  seed = 42L,
  calibration_ids = NULL,
  hold_out_calibration = FALSE,
  ...
)
```

Arguments

`meta`	Store metadata: needs x, y and sample_id.
`k`	Number of folds.
`predpoints`	Where the map will be predicted: a data frame with x and y, or an sf object. REQUIRED unless `modeldomain` is passed through `...` – the method has no meaning without it, so it is not defaulted.
`test_ids`	Frozen test sample_ids. Preferred over `hold_out_test` whenever several plans are to be compared on the same held-out data.
`hold_out_test`	Carve a test set by running kNNDM with k + 1 folds and holding one out. FALSE by default: a test set that changes with k is not a frozen test set.
`crs`	CRS of `meta$x`/`meta$y`. 4326 by default.
`project_to`	Projection used for the distance comparison, or NULL to use the coordinates as they are (correct only if already projected).
`seed`	Seed, for the sampling kNNDM does internally.
`calibration_ids`	Frozen calibration-set sample_ids, as `test_ids`.
`hold_out_calibration`	Carve a calibration set for the intervals as the test set is carved: kNNDM in k + 1 folds over what the test left, one held out. FALSE by default.
`...`	Passed to CAST::knndm() – `maxp`, `clustering`, `samplesize`, `modeldomain`, `space`.

Value

A fold_plan.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
elevation <- terra::rast(file.path(ex$raster_dir, "elevation.tif"))
knndm_folds(meta, k = 3, predpoints = prediction_sample(elevation, size = 500))
```

`latest_run_dir` *The most recent run under* `base`*, by time, and only if it finished.*

Description

WHY THIS IS NOT `sort(dirs, decreasing = TRUE)[1]`.

Usage

```
latest_run_dir(
  base,
  prefix,
  require_file = NULL,
  require_pattern = NULL,
  label = "run",
  on_none = c("stop", "null")
)
```

Arguments

`base`	directory holding the run directories.
`prefix`	run ids must start with this ("soc_", "final_"). "" accepts every directory.
`require_file`	path INSIDE a run that only exists when it finished, e.g. "comparison/comparison_ranked.csv". A run without it is not a candidate, and its mtime is what "newest" is measured on. NULL accepts any directory, which is almost never what you want.
`require_pattern`	alternative to require_file for runs whose completion is a FAMILY of files rather than one (05c's shard logs): a regex that at least one file directly inside the run must match. "Newest" is then the newest matching file, so a run still being written wins over an old one.
`label`	what to call these runs in messages.
`on_none`	what to do when no run qualifies. "stop" (default) for a stage that cannot proceed without one; "null" for a CHECK that must report "incomplete" and carry on – 99_check_pipeline.R does that, and a stop() there would turn a diagnosis into a crash. Returning NULL is loud only if the caller tests for it; every caller that passes "null" here prints a line saying so.

Details

That expression means "last alphabetically", which equals "most recent" only while every run id is a timestamp sharing one prefix. Runs given names broke it silently: `soc_0_5cm_design_spatial` sorts ahead of `soc_0_5cm_20260916_232318` because 'd' > '2', so "latest" started resolving to a run that had died on its first unit, and stage 04 would have refit the final model against it without a word.

It also never asked whether the run FINISHED. Stage 04 creates its output directory before its own validations run, so a failure leaves a `final_<timestamp>` behind that "latest" would then deploy – which is what B2's review found from the other direction.

Value

The run id (basename), or stop() / NULL when nothing qualifies.

Examples

```
base <- file.path(tempdir(), "runs")
for (id in c("run_a", "run_b")) {
  dir.create(file.path(base, id, "comparison"), recursive = TRUE, showWarnings = FALSE)
  writeLines("done", file.path(base, id, "comparison", "comparison_ranked.csv"))
}
Sys.setFileTime(file.path(base, "run_a", "comparison", "comparison_ranked.csv"),
                Sys.time() - 3600)
dir.create(file.path(base, "run_c"), showWarnings = FALSE)   # started, never finished
latest_run_dir(base, "run_", require_file = "comparison/comparison_ranked.csv")
```

---

list_models *Every registered model, as a table.*

Description

Every registered model, as a table.

Usage

```
list_models()
```

Value

A tibble: name, input ("table" or "patches"), tunable (whether it draws its own grid) and description.

Examples

```
list_models()
```

---

make_manual_tune_grid *Build a manual grid from explicit lists of values per parameter.*

Description

Build a manual grid from explicit lists of values per parameter.

Usage

```
make_manual_tune_grid(...)
```

Arguments

...	Named arguments, each a vector or list of values to cross. Only the provided parameters are varied; all others use their first (default) value from the parameter space.

Value

A tibble, one configuration per row, with a config_id column.

Examples

```
make_manual_tune_grid(
  embedding_dim = c(256L, 384L),
  gate_type     = c("vector_featurewise", "no_gate_concat"),
  base_lr       = c(0.001, 0.0005)
)
```

---

make_tune_grid *Generate a random hyperparameter grid (like caret's tuneLength).*

Description

Generate a random hyperparameter grid (like caret's tuneLength).

Usage

```
make_tune_grid(
  tune_length = 20L,
  seed = NULL,
  fixed = list(),
  windows = NULL,
  n_train = NULL
)
```

Arguments

tune_length	Number of configurations to sample.
seed	Random seed for reproducibility.
fixed	Named list used to FIX or RESTRICT parameters: • a length-1 value fixes the parameter (e.g. loss_fn = "smooth_l1") • a length>1 vector restricts the sampling pool (e.g. base_lr = c(1e-4, 3e-4)) • for the list-valued params window_sizes / conv_channels, pass a list of options (e.g. window_sizes = list(c(7L), c(5L, 7L))) or a single vector. Example focused search: fixed = list(loss_fn = "smooth_l1", batch_size = 512L, base_lr = c(1e-4, 3e-4), window_sizes = list(c(7L), c(5L, 7L)))

windows — The patch sizes the store holds (`data$store$window_sizes`). The window options become every single window and every pair of them – a dual-branch model takes two. NULL keeps the SOC example's 3/9/15 set; dsm_train() always passes the store's.

n_train — Training points in the smallest fold. Batch sizes that would give an epoch fewer than four gradient steps are left out; if none is left, the largest power of two that gives four is used. NULL keeps the full set. dsm_train() passes it from the plan.

Value

A tibble with one row per configuration. window_sizes and conv_channels are stored as list-columns.

Examples

```
make_tune_grid(tune_length = 3, seed = 1, windows = c(3, 7))
```

`model_spec` *Describe a model the framework can fit.*

Description

Describe a model the framework can fit.

Usage

```
model_spec(
  name,
  input,
  fit,
  predict,
  default_grid = NULL,
  count_params = NULL,
  description = "",
  fit_args = NULL
)
```

Arguments

name — Unique identifier, e.g. "rf", "mlp", "cnn".

input — "table" or "patches" – which view of the fold cache the model consumes.

fit — function(x, y, cfg, ...) returning a fitted object.

predict — function(object, x, ...) returning a numeric vector.

`default_grid` function(tune_length, seed, x, y) returning a tibble with a config_id column, or NULL if the model has nothing to tune.

x AND y ARE THE REAL TRAINING DATA OF THE FIRST FOLD, and they are in the signature because some grid generators need them. mtry is a fraction of ncol(x); glmnet's lambda path is computed from the VALUES. A grid built against a synthetic matrix of the right width is correct for the first kind and quietly wrong for the second, so the framework hands over the real thing and a generator takes what it needs.

A generator that needs neither still has to accept them – an argument it ignores costs nothing, and a signature that varies per model is a signature the runner cannot call.

A PATCH model's generator may also take `windows`, the patch sizes the loaded store holds, and `n_train`, the training points of the smallest fold. dsm_train() passes each one the generator declares, so a default grid never asks for a window the store does not have – the CNN's grid used to draw 3/9/15, the SOC example's, from any store – or for a batch no fold can fill.

`count_params` function(object) returning the number of free parameters, used as the complexity axis in one_se(). NULL means the model cannot report it and one_se() falls back to its other rule.

`description` One line, printed by list_models().

`fit_args` The arguments a caller may hand fit() through dsm_train()'s . . . . NULL derives them from fit()'s formals, less the ones the framework supplies itself (x, y, cfg, x_val, y_val, device, n_cores, points_valid, transform, model_name) and . . . . dsm_train() refuses any other: a misspelt option for a table model used to vanish into fit()'s . . . and train as if it had not been given.

Value

A model_spec.

Examples

```
mean_only <- model_spec(
  "mean_only", input = "table",
  fit = function(x, y, cfg, ...) list(mu = mean(y)),
  predict = function(object, x, ...) rep(object$mu, nrow(x)),
  description = "the training mean: a floor every model must beat")
mean_only
```

`occlusion_report` *What each part of the patch is worth to one trained unit of a tuning run.*

Description

Hides a part of every patch – the centre pixel, each ring around it, the whole context – and measures what the unit loses for it. The caller names the unit and nothing more: the checkpoint is found, the architecture rebuilt from the grid and the fold's cache from the plan.

Usage

```
occlusion_report(
  run_dir,
  data,
  config_id,
  fold = 1L,
  seed_i = 1L,
  role = "validation",
  transform = NULL,
  device,
  ...
)
```

Arguments

run_dir	Tuning run directory.
data	dsm_data, or a list with store, points, type_table.
config_id	Which config.
fold, seed_i	Which unit of it.
role	Which split to measure on. Validation by default: the test set is frozen, and occlusion is a diagnostic, not a result.
transform	NULL (the default) for the inverse of the transform the store was built under, as in dsm_train(); a function to use instead, refused if it disagrees with the store's.
device	torch device.
...	method: "permute" (the default), each hidden pixel takes another point's value – real terrain, somebody else's – or "zero"; seed, of the permutation; clamp, the plausible range of the target, c(0, Inf) by default.

Value

A spatial_occlusion: table, one row per part hidden, with the pixels hidden, the CCC and MAE without them and the change in CCC, and baseline_ccc, the unit's own. Printed with what it does and does not mean.

Examples

```
run <- example_run()    # a small fitted run, made once a session
occlusion_report(run$fit$run_dir, run$data, config_id = run$final$selected_config_ids,
                 device = torch::torch_device("cpu"))
```

one_se *Pick the simplest config within one standard error of the best.*

Description

Pick the simplest config within one standard error of the best.

Usage

```
one_se(by_config, metric = "val_ccc", complexity = "n_params", maximise = NULL)
```

Arguments

by_config	From summarise_resamples().
metric	Metric column stem, e.g. "val_ccc".
complexity	Column holding the simplicity ordering (lower = simpler), or a numeric vector the same length. Defaults to n_params, which count_model_params() produces.
maximise	TRUE when higher is better (CCC, R2); FALSE for an error.

Value

One row of by_config, with within_one_se (how many configs were tied) and simpler_than_best (whether the rule actually moved the choice) attached – a selection rule that silently returns the same answer as the default should say so.

Examples

```
# A tuning table: three configurations, each on three folds under two seeds
set.seed(1)
cfgs <- data.frame(config_id = c("cfg_001", "cfg_002", "cfg_003"),
                   skill = c(0.615, 0.62, 0.55), n_params = c(2e5, 2e6, 5e4))
cmp <- merge(expand.grid(config_id = cfgs$config_id, fold = 1:3, seed = 1:2,
                         stringsAsFactors = FALSE), cfgs)
cmp$status  <- "success"
cmp$val_ccc <- cmp$skill + rnorm(nrow(cmp), 0, 0.02)
pick <- one_se(summarise_resamples(cmp, metrics = "val_ccc"), metric = "val_ccc")
pick$config_id
```

`paired_family_test` *Paired comparison between two model families.*

Description

WHY PAIRED, AND WHY IT MATTERS HERE.

Usage

```
paired_family_test(
  a,
  b,
  metric = "val_ccc",
  config_a = NULL,
  config_b = NULL,
  label_a = "a",
  label_b = "b",
  conf = 0.95
)
```

Arguments

`a, b`	Comparison tibbles (or the `comparison` element of a fit).
`metric`	Column to compare. Higher-is-better is assumed for the verdict text unless the name matches a known error metric.
`config_a, config_b`	Which config of each family to compare. Defaults to the best by mean `metric`, which is the config someone would deploy.
`label_a, label_b`	Names for the report.
`conf`	Interval level.

Details

Every family in stage 03b runs on the SAME fold plan with the SAME seeds, by construction. So each family's units come in matched pairs: rf_context on fold 2 seed 3 and the CNN on fold 2 seed 3 saw the identical training rows.

Comparing the two MEANS and their separate spreads throws that away. Most of the spread across units is the fold – some folds are simply harder, and both families suffer on them together. A paired comparison subtracts that shared difficulty out before asking whether anything is left.

The first 03b run made exactly this mistake in the reader's favour: it reported +0.0091 CCC against a seed noise floor of 0.0383 and concluded "smaller than the noise". The conclusion happened to hold, but the reasoning did not – the noise floor is the spread of ONE family across seeds, which is not the standard error of a DIFFERENCE, and comparing a difference to it can hide a real effect as easily as it can invent one.

What is reported is a mean difference with its own standard error and a paired t interval. Not a p-value on its own: with 9 pairs, "not significant" is mostly a statement about 9, and the interval says how large an effect is still compatible with the data – which is the actual question when the answer is "the convolution buys nothing".

Value

An object of class "paired_comparison".

Examples

```
# A tuning table: three configurations, each on three folds under two seeds
set.seed(1)
cfgs <- data.frame(config_id = c("cfg_001", "cfg_002", "cfg_003"),
                   skill = c(0.615, 0.62, 0.55), n_params = c(2e5, 2e6, 5e4))
cmp <- merge(expand.grid(config_id = cfgs$config_id, fold = 1:3, seed = 1:2,
                         stringsAsFactors = FALSE), cfgs)
cmp$status  <- "success"
cmp$val_ccc <- cmp$skill + rnorm(nrow(cmp), 0, 0.02)
# a second family, trained on the same folds under the same seeds
forest <- cmp
forest$config_id <- sub("cfg", "rf", forest$config_id)
forest$val_ccc <- forest$val_ccc - 0.03 + rnorm(nrow(forest), 0, 0.01)
paired_family_test(cmp, forest, metric = "val_ccc", config_a = "cfg_002",
                   config_b = "rf_002", label_a = "cnn", label_b = "forest")
```

---

permutation_importance

Permutation importance: how much a fitted model relies on each variable.

Description

Each point takes one variable's whole patch – every pixel, every window – from another point; the model predicts again, and the loss of skill is the variable's importance to that model. Nothing is retrained.

Usage

```
permutation_importance(
  draws = 5L,
  within = NULL,
  fill = c("permute", "mean"),
  metric = c("ccc", "rmse", "rmse_transform"),
  seed = 42L
)
```

Arguments

`draws`	Permutations per variable and per model. The importance is their mean; their spread is reported beside the spread between models.
`within`	Where a donor may come from. NULL (the default): any point of the rows scored. A number: square blocks of that side, in the coordinate units of the store's `x` and `y` – the importance beyond the region. A column name of the point table (an ecoregion, a soil region): the same class. Or a vector with one label per point of the store. A point alone in its block or class keeps its own values; how many did is reported.
`fill`	"permute" (the default), or "mean": the variable is set to its training mean everywhere in the patch – an ablation, which puts the patch off the data (see Details) and is read against the permutation.
`metric`	What the table is ranked by: "ccc" (the drop in Lin's concordance, native units), "rmse" (the rise in RMSE, native units) or "rmse_transform" (the rise in RMSE in the space the model was trained in, e.g. log1p). All three are computed and kept.
`seed`	Seed of the permutations. The same draws are given to every variable and every model.

Details

A variable whose permutation costs nothing is one the model does not use. It may still carry information other variables also carry: permutation measures reliance, not necessity, and a model that can find the same thing elsewhere leans on whichever it found first.

The mean of a z-scored channel is 0 and of a dummy its class frequency, so `fill = "mean"` gives every point the average landscape, flat. A large drop under it with a small one under permutation says the model reacts to the input being unusual rather than to its content.

Value

An `importance_spec`, for `dsm_importance()`.

Examples

```
permutation_importance()
permutation_importance(draws = 3, within = 0.05)   # donors from the same 0.05 block
```

---

picp_report — *Coverage, and whether it holds where it is needed.*

Description

Coverage, and whether it holds where it is needed.

Usage

```
picp_report(obs, lower, upper, group = NULL, alpha = 0.1)
```

Arguments

obs, lower, upper	As in picp().
group	Optional grouping (a spatial block, a region, a soil class).
alpha	The nominal miscoverage, for the verdict.

Value

An object of class "picp_report".

Examples

```
set.seed(1)
obs  <- rlnorm(300, 3, 0.4)
pred <- obs * exp(rnorm(300, 0, 0.25))
cal  <- conformal_calibrate(obs[1:150], pred[1:150], alpha = 0.1)
iv <- conformal_interval(cal, pred[151:300], lower_limit = 0)
# one width for every level: right on average, and by level?
picp_report(obs[151:300], iv$lower, iv$upper, alpha = 0.1,
            group = ifelse(pred[151:300] > median(pred), "high", "low"))
```

---

plot.block_bootstrap_by_size

Plot the interval as the blocks grow.

Description

Plot the interval as the blocks grow.

Usage

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

Arguments

x	A block_bootstrap_by_size, from block_bootstrap_by_size().
...	Ignored.

Value

x, invisibly.

plot.dsm_importance *Plot a variable importance.*

Description

The figure that answers the importance's own question: bars for a permutation, the cost per pixel by ring (or per variable, or per window) for a context, the summary plot for SHAP – a point per profile at its SHAP value, coloured by the variable's value there – the curves for ALE, for SAGE the loss each variable takes away, and for a refit the skill a network trained without it loses.

Usage

```
## S3 method for class 'dsm_importance'
plot(x, n = 20L, ...)
```

Arguments

x	A dsm_importance, from dsm_importance().
n	How many variables to show.
...	Ignored.

Value

x, invisibly.

plot.importance_comparison *Plot importances side by side.*

Description

The first two importances' shares of their largest, one against the other, each variable a point on a 1:1 line; and, for two SHAP importances of the same points, how well they agree variable by variable.

Usage

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

Arguments

x	An importance_comparison, from compare_importance().
...	Ignored.

Value

x, invisibly.

---

plot.importance_map *Plot maps of SHAP values.*

Description

One panel per variable on a diverging scale centred on zero – where the variable raises the prediction and where it lowers it – then the dominant variable and the prediction.

Usage

```
## S3 method for class 'importance_map'
plot(x, n = 12L, ...)
```

Arguments

x	An importance_map, from importance_map().
n	How many variables, the most important first.
...	Ignored.

Value

x, invisibly.

---

plot.spatial_correlogram *Plot a spatial correlogram.*

Description

Plot a spatial correlogram.

Usage

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

Arguments

x	A spatial_correlogram, from spatial_correlogram().
...	Ignored.

Value

x, invisibly.

prediction_sample *A regular sample of the prediction area, as kNNDM's* `predpoints`.

Description

kNNDM needs the places the map will cover, not the places it was trained on. A few thousand regularly spaced pixels describe that well enough – the method compares DISTRIBUTIONS of distances, and a regular sample estimates one cheaply. A random sample would work too and is noisier for the same size.

Usage

```
prediction_sample(raster, size = 5000L)
```

Arguments

`raster`	A SpatRaster (terra) covering the prediction area.
`size`	About how many points to draw. Cells where the raster is NA – the sea around a continent – do not count: when too few land on data, the sample is drawn again, denser.

Value

A data frame with x and y, in the raster's own CRS.

Examples

```
ex <- example_landscape()
elevation <- terra::rast(file.path(ex$raster_dir, "elevation.tif"))
head(prediction_sample(elevation, size = 200))
```

print_aoa *Report what an AOA covers.*

Description

Prints the threshold, the share of cells inside it and the quantiles of the DI, and says so plainly when less than half the map is inside.

Usage

```
print_aoa(di, threshold, label = "prediction area")
```

Arguments

di	Dissimilarity index of the cells, from dissimilarity_index().
threshold	From aoa_threshold().
label	What the cells are, for the report.

Value

The logical mask of the cells inside (inside_aoa()), invisibly.

Examples

```
set.seed(1)
x_train <- matrix(rnorm(200), 100, 2)    # 100 training rows of 2 scaled predictors
ref <- di_reference(x_train)
threshold <- aoa_threshold(ref, folds = rep(1:5, 20))
print_aoa(dissimilarity_index(ref, matrix(rnorm(400, sd = 1.5), 200, 2)), threshold)
```

---

print_noise_floor *Print a noise-floor report in the terms it should be read in.*

Description

Print a noise-floor report in the terms it should be read in.

Usage

```
print_noise_floor(nf, digits = 4L)
```

Arguments

nf	From seed_noise_floor().
digits	Digits of the numbers.

Value

nf, invisibly; NULL when no floor could be estimated.

Examples

```
# A tuning table: three configurations, each on three folds under two seeds
set.seed(1)
cfgs <- data.frame(config_id = c("cfg_001", "cfg_002", "cfg_003"),
                   skill = c(0.615, 0.62, 0.55), n_params = c(2e5, 2e6, 5e4))
cmp <- merge(expand.grid(config_id = cfgs$config_id, fold = 1:3, seed = 1:2,
                         stringsAsFactors = FALSE), cfgs)
cmp$status  <- "success"
cmp$val_ccc <- cmp$skill + rnorm(nrow(cmp), 0, 0.02)
print_noise_floor(seed_noise_floor(cmp, metric = "val_ccc"))
```

print_one_se — *Say what one_se() did, including when it did nothing.*

Description

Say what one_se() did, including when it did nothing.

Usage

```
print_one_se(pick, metric = "val_ccc", digits = 4L)
```

Arguments

pick	From one_se().
metric	The metric it selected on, for the report.
digits	Digits of the threshold.

Value

pick, invisibly.

Examples

```
# A tuning table: three configurations, each on three folds under two seeds
set.seed(1)
cfgs <- data.frame(config_id = c("cfg_001", "cfg_002", "cfg_003"),
                   skill = c(0.615, 0.62, 0.55), n_params = c(2e5, 2e6, 5e4))
cmp <- merge(expand.grid(config_id = cfgs$config_id, fold = 1:3, seed = 1:2,
                         stringsAsFactors = FALSE), cfgs)
cmp$status  <- "success"
cmp$val_ccc <- cmp$skill + rnorm(nrow(cmp), 0, 0.02)
print_one_se(one_se(summarise_resamples(cmp, metrics = "val_ccc"), metric = "val_ccc"))
```

random_cv — *Random k-fold. Ignores geography by construction.*

Description

Right when the rows really are independent, and the cleanest way to MEASURE what geography is worth: run it against spatial_cv() on the same points and the gap is the spatial optimism.

Usage

```
random_cv(
  k = 5L,
  test_frac = 0.15,
  group = "auto",
  seed = 42L,
  calibration_frac = 0
)
```

Arguments

`k`	Number of folds.
`test_frac`	Share held out as the test set, drawn first.
`group`	"auto" keeps the rows of one profile_id together when the table repeats profiles; NULL or "row" makes each row its own unit; a column name, or a vector with one label per row, names the groups.
`seed`	Seed for the partition only (see with_local_seed): a fixed plan reproduces even when the training seeds change.
`calibration_frac`	Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

Value

A `resample_spec`, as spatial_cv() returns.

Examples

```
random_cv(k = 10)
```

`random_folds` *Random k-fold over the train+validation pool.*

Description

Ignores geography by construction. On spatially clustered data that makes the validation metric optimistic – the pipeline check measures it: a random split put ~27% of validation in the SAME 250 m pixel as a training point. Offered anyway, for two reasons: it is the right answer when the rows really are independent, and running it against spatial_folds() on the same data is the cleanest way to MEASURE how much geography is worth here.

Usage

```
random_folds(
  meta,
  k = 5L,
  test_frac = 0,
  test_ids = NULL,
  seed = 42L,
  group = "auto",
  calibration_frac = 0,
  calibration_ids = NULL
)
```

Arguments

`meta` Patch store meta.

`k` Number of folds.

`test_frac` Share held out as the test set, drawn first.

`test_ids` Sample ids that ARE the test set, in place of `test_frac`: a test set drawn again on every run is not a test set.

`seed` Seed for the partition only (see with_local_seed): a fixed plan reproduces even when the training seeds change.

`group` "auto" keeps the rows of one profile_id together when the table repeats profiles; NULL or "row" makes each row its own unit; a column name, or a vector with one label per row, names the groups.

`calibration_frac` Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

`calibration_ids` Sample ids that ARE the calibration set, in place of `calibration_frac`, as `test_ids` is for the test set.

Value

A `fold_plan` with `k` folds.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
random_folds(meta, k = 3, test_frac = 0.2)
```

refit_importance *Refit importance: what the model loses when trained without a variable.*

Description

Leave one covariate out (LOCO; Lei et al. 2018, Williamson et al. 2023): each variable's channels are set to their training mean – every pixel, every window, every row – and the final run's seeds are trained again, everything else as the run trained them: its split, scaling, schedule, thread count, and the seed. The test skill lost is the variable's importance. A permutation asks how much the fitted network relies on a variable; this asks whether the variable is needed at all: variables that carry the same information each cost little, because a network trained without one finds it in the other.

Usage

```
refit_importance(
  metric = c("ccc", "rmse", "rmse_transform"),
  check_seeds = 1L,
  run_id = NULL,
  resume = TRUE,
  output_dir = NULL,
  n_cores = NULL,
  max_ram_gb = NULL
)
```

Arguments

metric	What the table is ranked by: "ccc" (the drop in Lin's concordance, native units), "rmse" (the rise in RMSE, native units) or "rmse_transform" (in the space the model was trained in). All three are kept.
check_seeds	How many of the seeds are trained again with nothing left out first, which must reproduce the run. 1 or more: a seed's numbers depend only on its seed and its thread count, so one checks the path.
run_id	NULL for `refit_<timestamp>`. An existing one, with resume = TRUE, finishes a refit that stopped: its finished units are kept.
resume	Keep the units an existing run_id already trained.
output_dir	Where refits go. NULL: refit_importance/ in the final run's directory.
n_cores	Cores for the refit; NULL is the physical cores minus one. Each unit trains with the final run's threads per unit – part of its numbers – so n_cores must be at least that.
max_ram_gb	RAM the workers may use in total, as in `dsm_final()`.

Details

Before any variable is left out, the first check_seeds seeds are trained again with nothing left out, and must give the run's own predictions – or the refit is not the run's training, and it stops. It trains (variables x seeds + check_seeds) networks, side by side as dsm_final() does: give dsm_importance() themes as groups, and a few of the run's seeds.

Value

An importance_spec, for dsm_importance().

Examples

```
refit_importance(check_seeds = 1)
```

---

region_cv — *Leave-region-out, on a grouping that already exists (biome, catchment, ...).*

Description

Leave-region-out, on a grouping that already exists (biome, catchment, ...).

Usage

```
region_cv(group, k = NULL, test_frac = 0.15, seed = 42L, calibration_frac = 0)
```

Arguments

group	Group labels, one per row of meta.
k	Number of folds; defaults to one per group.
test_frac	Share held out as the test set, drawn first.
seed	Seed for this split only; the training seeds do not move it.
calibration_frac	Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

Value

A resample_spec, as spatial_cv() returns.

Examples

```
ex <- example_landscape()
region_cv(group = ex$profiles$survey)
```

`region_folds` *Fold by a grouping column: no group is split across folds.*

Description

For when the unit that must not leak is named rather than geometric – a survey campaign, a country, a laboratory, a soil map unit. With k = number of groups this is leave-one-group-out.

Usage

```
region_folds(
  meta,
  group,
  k = NULL,
  test_frac = 0,
  test_ids = NULL,
  seed = 42L,
  calibration_frac = 0,
  calibration_ids = NULL
)
```

Arguments

`meta`	Patch store meta.
`group`	Group labels, one per row of `meta`.
`k`	Number of folds; defaults to one per group.
`test_frac`	Share held out as the test set, drawn first.
`test_ids`	Sample ids that ARE the test set, in place of `test_frac`: a test set drawn again on every run is not a test set.
`seed`	Seed for this split only; the training seeds do not move it.
`calibration_frac`	Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.
`calibration_ids`	Sample ids that ARE the calibration set, in place of `calibration_frac`, as `test_ids` is for the test set.

Value

A `fold_plan`.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
region_folds(meta, group = ex$profiles$survey, k = 3)
```

---

register_model *Add a model to the registry.*

Description

Add a model to the registry.

Usage

```
register_model(spec, overwrite = FALSE)
```

Arguments

spec	From model_spec().
overwrite	FALSE (default) refuses to replace an existing name. A silent replacement is how two different models come to answer to the same name in the same session, and the results carry no mark of which ran.

Value

spec, invisibly.

Examples

```
mean_only <- model_spec(
  "mean_only", input = "table",
  fit = function(x, y, cfg, ...) list(mu = mean(y)),
  predict = function(object, x, ...) rep(object$mu, nrow(x)),
  description = "the training mean: a floor every model must beat")
register_model(mean_only, overwrite = TRUE)
list_models()      # dsm_train(data, model = "mean_only", ...) now fits it
```

`resolve_resampling` *Turn a spec into a fold plan against real points.*

Description

Exported because a plan is worth looking at before spending a night on it: `plan <- resolve_resampling(spatial_cv(k = 5`

Usage

```
resolve_resampling(
  spec,
  data,
  test_ids = NULL,
  windows = NULL,
  verbose = TRUE,
  calibration_ids = NULL
)
```

Arguments

`spec`	From spatial_cv() and friends, or an existing fold_plan, which is returned unchanged.
`data`	From dsm_load().
`test_ids`	Sample ids to force into the test set, so a frozen test set survives a change of method.
`windows`	Windows the grid will use, for `buffer = "auto"`.
`verbose`	Print the block size that `block_size = "auto"` chose.
`calibration_ids`	Sample ids to force into the calibration set, in place of the `calibration_frac` of `spec`: several designs then calibrate their "split" intervals on the same points.

Value

A `fold_plan`, already checked with check_fold_plan().

Examples

```
ex <- example_landscape()
store <- dsm_prepare(ex$profiles, target = "soc_stock", raster_dir = ex$raster_dir,
                     windows = c(3, 7), out_dir = file.path(tempdir(), "landscape"),
                     percentage = "^clay_pct$", transform = "log1p", n_cores = 1,
                     overwrite = TRUE, verbose = FALSE)
data <- dsm_load(store, windows = integer(0), verbose = FALSE)
resolve_resampling(spatial_cv(k = 3, block_size = 0.05, test_frac = 0.2), data,
                   windows = c(3, 7))
```

sage_importance *SAGE: how much of the model's skill each variable carries, shared fairly.*

Description

Shapley Additive Global importancE (Covert, Lundberg & Lee 2020): the Shapley values of the loss the model explains. The value of a set of variables is the loss of the model's predictions when only they are known – the others taken from background points of the model's own training data, the whole patch, the same point in every window, as in `shap_importance()("kernel")`. A variable's SAGE value is the loss it takes away; variables that carry the same information split it, where a permutation credits the shared part to neither and a refit without one gives each nothing (the other stands in). The values add up exactly to the loss of the mean prediction – no variable known – less the model's own.

Usage

```
sage_importance(
  loss = c("mse", "mae"),
  background = 16L,
  max_points = 200L,
  permutations = 64L,
  seed = 42L
)
```

Arguments

`loss`	"mse" (the default) or "mae", in the space the model was trained in (log1p for a log1p target).
`background`	How many of the model's training points the absent variables are drawn from; each coalition costs one pass per point each.
`max_points`	How many of the rows the loss is taken over, drawn by the seed – the kernel's default, so the two read the same points at the same cost. More points, a steadier loss, and a longer run.
`permutations`	Above 14 variables: permutations sampled, in pairs.
`seed`	Seed of the draws, shared by every model.

Details

Exact over every coalition up to 14 variables, by sampled permutations up to 40: meant for a few variables – give `groups` themes.

Value

An `importance_spec`, for `dsm_importance()`.

Examples

```
sage_importance(background = 8, max_points = 100)
```

---

score_test_grid *Score every trained unit of a tuning run on the held-out test set.*

Description

Score every trained unit of a tuning run on the held-out test set.

Usage

```
score_test_grid(
  run_dir,
  data,
  transform = NULL,
  device,
  config_ids = NULL,
  allow_unfrozen = FALSE
)
```

Arguments

run_dir	Tuning run directory.
data	A dsm_data (from dsm_load()), or a list with store, points and type_table.
transform	NULL (the default) for the inverse of the transform the store was built under, as in dsm_train(); a function is the inverse to use instead, and is refused if it disagrees with the store's.
device	torch device.
config_ids	Which configs to score. NULL means every config in the grid.
allow_unfrozen	Escape hatch for teaching or for a run whose selection was recorded elsewhere. Not a default, and the report says it was used.

Value

An object of class "test_optimism".

Examples

```
run <- example_run()    # a small fitted run, made once a session
# every configuration on the test set, once the choice is frozen
score_test_grid(run$fit$run_dir, run$data, device = torch::torch_device("cpu"))
```

seed_noise_floor *How much does a metric move when ONLY the seed changes?*

Description

The noise floor of the whole experiment. Computed within (config, fold), so nothing but the random draw differs between the runs being compared; then summarised across all configs.

Usage

```
seed_noise_floor(comparison, metric = "val_ccc")
```

Arguments

comparison	Unit-level comparison table.
metric	Metric column to measure.

Details

Any difference between two configs smaller than this is not evidence. That sentence is the entire reason the function exists, and it is why oneSE-style selection is worth having: the best mean is frequently the luckiest draw.

Value

A list: metric; by_config, one row per (config, fold) trained under more than one seed, with the mean, sd and range over the seeds; n_comparable, how many such rows; median_sd, the floor; and max_range, the widest spread seen.

Examples

```
# A tuning table: three configurations, each on three folds under two seeds
set.seed(1)
cfgs <- data.frame(config_id = c("cfg_001", "cfg_002", "cfg_003"),
                   skill = c(0.615, 0.62, 0.55), n_params = c(2e5, 2e6, 5e4))
cmp <- merge(expand.grid(config_id = cfgs$config_id, fold = 1:3, seed = 1:2,
                         stringsAsFactors = FALSE), cfgs)
cmp$status  <- "success"
cmp$val_ccc <- cmp$skill + rnorm(nrow(cmp), 0, 0.02)
nf <- seed_noise_floor(cmp, metric = "val_ccc")
nf$median_sd
```

selected_config_id *The configuration a final run deployed.*

Description

Read from the list of configurations in the order they were chosen – not from the grid's order, in which a runner-up can come first.

Usage

```
selected_config_id(summary, label = "this final run")
```

Arguments

summary	the list read from comparison/final_run_summary.rds.
label	what to call the run in messages.

Value

the config id, or stop() – "auto" must never travel on unresolved.

Examples

```
summary <- list(selected_config_ids = c("cfg_004", "cfg_011"))
selected_config_id(summary)   # the first chosen; the grid's order can differ
```

set_torch_threads *Set torch's thread pools.*

Description

Split out of setup_torch_device() so that a caller who already holds a device can change the threads without building a second one – dsm_train(device = d, n_cores = 8) is that caller. The threads belong to the R session, not to a device: whatever set them last is what every later torch call gets.

Usage

```
set_torch_threads(n_threads = NULL)
```

Arguments

n_threads	NULL for the physical cores minus one (see resolve_cores()), or a whole number >= 1.

Details

The interop pool can be sized once per session, before torch first runs work in parallel; later calls leave it as it is and say so once.

Value

The number of threads, invisibly.

Examples

```
before <- Sys.getenv(c("OMP_NUM_THREADS", "MKL_NUM_THREADS"), unset = NA)
n <- torch::torch_get_num_threads()
set_torch_threads(2)
torch::torch_get_num_threads()
# the session as it was: torch's threads, and the two variables set here
torch::torch_set_num_threads(n)
Sys.unsetenv(names(before)[is.na(before)])
if (any(!is.na(before))) do.call(Sys.setenv, as.list(before[!is.na(before)]))
```

---

setup_torch_device *Configure torch threads and select compute device.*

Description

Configure torch threads and select compute device.

Usage

```
setup_torch_device(n_threads = NULL, use_cuda = TRUE)
```

Arguments

n_threads	Number of intra-op threads; NULL for the physical cores minus one (see resolve_cores()).
use_cuda	Use GPU if available.

Value

A torch_device object.

Examples

```
before <- Sys.getenv(c("OMP_NUM_THREADS", "MKL_NUM_THREADS"), unset = NA)
n <- torch::torch_get_num_threads()
device <- setup_torch_device(n_threads = 2, use_cuda = FALSE)
device
# the session as it was: torch's threads, and the two variables set here
torch::torch_set_num_threads(n)
Sys.unsetenv(names(before)[is.na(before)])
if (any(!is.na(before))) do.call(Sys.setenv, as.list(before[!is.na(before)]))
```

`shap_importance` *SHAP importance: how each variable pushes each prediction, up or down.*

Description

Shapley values (Lundberg & Lee 2017) of the network's predictions, estimated from its gradients. Each point's prediction, less a reference prediction, is shared among the variables, with a sign: this one raised it, that one lowered it. Averaged in absolute value over the points they are a global importance – of how much each variable MOVES the predictions, which is not how much it improves them (`permutation_importance()` measures that); kept point by point (`$points`) they are the map of what drives the prediction where (Padarian et al. 2020; Wadoux et al. 2023 for soil carbon).

Usage

```
shap_importance(
  estimator = c("expected_gradients", "integrated_gradients", "kernel"),
  samples = 100L,
  background = NULL,
  steps = 50L,
  permutations = 64L,
  max_points = NULL,
  seed = 42L
)
```

Arguments

`estimator` "expected_gradients" (the default; Erion et al. 2021): the reference is a background of the model's own training points, drawn at random, so no baseline is chosen by hand – the values are SHAP values of the network, and the same estimator as SHAP's GradientExplainer. "integrated_gradients" (Sundararajan et al. 2017): one baseline, the training mean of every channel – the average landscape, flat, which no point is – kept because its sum is exact rather than sampled. "kernel": the Shapley values of the variables themselves, computed from coalitions, with no gradient (Lundberg & Lee 2017's KernelSHAP game): the value of a set of variables is the mean prediction, over a background of training points, with the variables outside it taken from the background point – the whole patch, the

same point in every window. Exact over every coalition up to 14 variables; by sampled permutations above, up to 40. Meant for a few variables – give `groups` themes – and a few points. Read against the gradient estimators: they share the total and split it alike when the variables do not interact, so where they part is how much the variables do.

`samples` For expected gradients: draws per point, each a background point and a place on the straight path to it. The noise they leave is reported.

`background` How many of the model's training points the references are drawn from. NULL: 200 for expected gradients, 16 for the kernel – each coalition costs one pass per background point.

`steps` For integrated gradients: points on the path.

`permutations` For the kernel above 14 variables: permutations sampled (in pairs, each with its reverse).

`max_points` Explain this many of the rows only, drawn at random – the same rows for every estimator, so two can be compared point by point. NULL: every row for the gradient estimators, 200 for the kernel.

`seed` Seed of the draws, shared by every model.

Details

The values are in the units the network predicts in: for a log1p target, a SHAP value of +0.1 raises the predicted value by about 10%. Each point's values add up to its prediction less the background's mean prediction (expected gradients) or less the prediction at the baseline (integrated gradients); `dsm_importance()` checks that they do and stops if they do not. SHAP by deep-network rules (DeepSHAP) is not offered: it needs a propagation rule written for every layer of this architecture.

Value

An `importance_spec`, for `dsm_importance()`.

Examples

```
shap_importance()
shap_importance("kernel", background = 8, max_points = 50)
```

`smear` *The conditional MEAN surface, from transform-space predictions.*

Description

The conditional MEAN surface, from transform-space predictions.

Usage

```
smear(
  pred_transform,
  cal,
  lower_limit = 0,
  method = c("global", "level", "total")
)
```

Arguments

`pred_transform`	Predictions in transform space.
`cal`	A smearing_cal, or a bare positive number to use as the factor.
`lower_limit`	Floor, e.g. 0 for a stock. Applied after the correction.
`method`	Which factor: "global" (the default; Duan's scalar), "level" (S as a function of the prediction, from bins of it, interpolated) or "total" (the exp(f)-weighted scalar, which unbiases a sum). The last two need a smearing_cal. `smearing_check()` measures them on held-out points.

Value

Numeric, in native units: an estimate of `E[y | x]` rather than of its median.

Examples

```
set.seed(1)
f <- runif(500, 2, 5)                    # predictions, in log1p space
z <- f + rnorm(500, 0, 0.4)              # what was observed, in log1p space
cal <- smearing_factor(z, f)
expm1(c(2, 3, 4))                          # the median surface
smear(c(2, 3, 4), cal)                     # the mean surface, by Duan's factor
smear(c(2, 3, 4), cal, method = "level")   # by a factor that follows the level
```

---

smear_map — *The mean surface of a median map already made.*

Description

The conditional mean is `exp(z) S(z) - 1` of the median `z` in transform space, so a mean map by any of `smear()`'s factors comes from the median map alone, cell by cell, without running the network again.

Usage

```
smear_map(
  median,
  cal,
  method = c("global", "level", "total"),
  lower_limit = 0,
  filename = ""
)
```

Arguments

median	The ensemble median, in native units: a SpatRaster, or the path of one – dsm_predict()'s ensemble_median band.
cal	A smearing_cal, from smearing_factor() or smearing_from_run().
method	As in smear().
lower_limit	As in smear().
filename	Where to write the map; "" keeps it in memory or in terra's temporary files.

Value

A SpatRaster: the conditional mean, cell by cell.

Examples

```
set.seed(1)
f <- runif(500, 2, 5)                    # predictions, in log1p space
z <- f + rnorm(500, 0, 0.4)              # what was observed, in log1p space
cal <- smearing_factor(z, f)
median_map <- terra::rast(nrows = 10, ncols = 10, vals = expm1(runif(100, 2, 5)))
smear_map(median_map, cal)
```

---

smearing_check — *The smearing factors on held-out points: which one unbiases the mean surface.*

Description

Each factor of smear() applied to held-out predictions – points none of them was calibrated on, a test set – against what was observed there: the bias of the total, which a total taken from the map inherits, and the bias within quintiles of the prediction, where a factor by level should help. The median surface, with no factor, is the first row. On the global SOC model, calibrated on three folds, the global factor won (+2.7% against -3.9% by level and -5.8% for a total), because those folds saw half the points the deployed model saw; this measures it again for a run of yours.

Usage

```
smearing_check(
  cal,
  obs_transform,
  pred_transform,
  methods = c("global", "level", "total"),
  lower_limit = 0
)
```

Arguments

`cal`	A smearing_cal, from `smearing_factor()` or `smearing_from_run()`.
`obs_transform, pred_transform`	Observed and predicted at held-out points, in transform space.
`methods`	The factors to compare.
`lower_limit`	As in `smear()`.

Value

A `smearing_check`: a tibble, one row per surface, with `bias_pct` (its total against the observed total, in %), `mae`, `rmse`, and `bias_pct_q1` to `bias_pct_q5` (within quintiles of the prediction, low to high).

Examples

```
set.seed(1)
f <- runif(1000, 2, 5)
z <- f + rnorm(1000, 0, 0.2 + 0.1 * (f - 2))   # the spread grows with the level
cal <- smearing_factor(z[1:500], f[1:500])
smearing_check(cal, z[501:1000], f[501:1000])  # on points it was not calibrated on
```

---

smearing_factor *The smearing factor, from held-out residuals in transform space.*

Description

The smearing factor, from held-out residuals in transform space.

Usage

```
smearing_factor(
  obs_transform,
  pred_transform,
  transform = c("log1p", "log"),
  bins = 5L
)
```

Arguments

`obs_transform`, `pred_transform`
: Observed and predicted values IN TRANSFORM SPACE (log1p here), on data the model did not train on. Training residuals are smaller than the ones new points will produce, so a factor calibrated on them under-corrects by exactly the amount the model overfits. ONE ROW PER POINT: see smearing_from_run() for why seeds are collapsed first.

`transform`
: Name of the forward transform. Only "log1p" and "log" are supported, and an unknown one is refused rather than assumed: the algebra above is specific to an exponential back-transform, and applying it to a square root or an identity would scale a number that needs no scaling.

`bins`
: Bins of the prediction for the factor by level (`smear(method = "level")`), cut at its quantiles; each needs 50 residuals.

Value

An object of class "smearing_cal".

Examples

```
set.seed(1)
f <- runif(500, 2, 5)                    # predictions, in log1p space
z <- f + rnorm(500, 0, 0.4)              # what was observed, in log1p space
cal <- smearing_factor(z, f)
cal
```

`smearing_from_run` *The smearing factor from a tuning run's out-of-fold predictions.*

Description

ONE RESIDUAL PER POINT, not one per (point, seed).

Usage

```
smearing_from_run(run_dir, config_id, role = "validation", transform = "log1p")
```

Arguments

`run_dir`
: Tuning run directory.

`config_id`
: Which config's residuals.

`role`
: Which role. "validation" is the point.

`transform`
: Name of the forward transform. Only "log1p" and "log" are supported, and an unknown one is refused rather than assumed: the algebra above is specific to an exponential back-transform, and applying it to a square root or an identity would scale a number that needs no scaling.

Details

The deployed prediction is the ensemble MEDIAN over seeds, so the residual being calibrated has to be the ensemble's and not one member's. This is the same reasoning cv_residuals() applies for the conformal interval, and for a while this function claimed to use "the same out-of-fold residuals" while quietly reading the per-seed rows instead: 9,276 rows for 3,092 points.

Two consequences, and the second is the one that matters:

- exp() is convex, so mean(exp(e)) grows with var(e). A single member's residual is noisier than the ensemble's (sd 0.6698 against 0.6507 here), which inflated S by 0.99% – almost exactly the 1.00% that the variance difference alone predicts.
- `n` read 9,276 when there were 3,092 exchangeable units, overstating the evidence threefold in the one line a reader would use to judge it.

The 1% is NOT why this was changed. The target is bracketed between 1.259 and 1.389 by two held-out measurements of the deployed ensemble itself, so no choice inside that range is defensible as more accurate. It was changed because the claim in the docs was false and the printed n was wrong.

Value

A smearing_cal, or NULL when the run wrote no usable predictions.

Examples

```
run <- example_run()    # a small fitted run, made once a session
smearing_from_run(run$fit$run_dir, run$final$selected_config_ids)
```

`spatial_correlogram` *Moran's I by distance class: how far apart two points stop resembling each other.*

Description

For each class of distance, Moran's I of `value` over the pairs of points that far apart: the mean product of their deviations from the mean, over the variance. Near zero, points that far apart are as unlike as any two; the distance where it falls to zero is the scale blocks should exceed. For two models compared at the same points, `value` is the difference of their errors – the quantity whose mean is compared.

Usage

```
spatial_correlogram(
  x,
  y,
  value,
  breaks_km = c(0, 5, 10, 25, 50, 100, 200, 400, 800),
  coords = c("lonlat", "metres")
)
```

Arguments

x, y	Coordinates: longitude and latitude in degrees (great-circle distances) or metres (straight-line distances), as coords says.
value	One number per point.
breaks_km	The edges of the distance classes, in km.
coords	"lonlat" (the default) or "metres".

Value

A spatial_correlogram: a tibble with from_km, to_km, n_pairs, moran_i, and expected, Moran's I with no spatial structure (-1/(n-1)).

Examples

```
ex <- example_landscape()
obs <- ex$profiles$soc_stock
set.seed(1)
pred_a <- obs * exp(rnorm(length(obs), 0, 0.20))   # two models' predictions
pred_b <- obs * exp(rnorm(length(obs), 0, 0.25))
d <- abs(pred_a - obs) - abs(pred_b - obs)    # what a comparison of the two compares
cg <- spatial_correlogram(ex$profiles$x, ex$profiles$y, d,
                          breaks_km = c(0, 1, 2, 5, 10, 20))
cg
plot(cg)
```

---

spatial_cv *Spatially blocked k-fold: whole blocks of ground go to one fold.*

Description

Spatially blocked k-fold: whole blocks of ground go to one fold.

Usage

```
spatial_cv(
  k = 5L,
  block_size = "auto",
  buffer = "auto",
  test_frac = 0.15,
  max_share = 0.1,
  buffer_metric = c("chebyshev", "euclidean"),
  seed = 42L,
  calibration_frac = 0
)
```

Arguments

`k` Folds.

`block_size` "auto" measures it from the points (see suggest_block_size); a number is used as given, in x/y units.

`buffer` "auto" is `max(window) * cell_size`, which is the exact distance at which two patches stop sharing a pixel under the Chebyshev metric; a number is used as given; NULL applies none.

`test_frac` Share held out entirely, carved by the same criterion.

`max_share` Balance constraint for "auto": the largest share of the points one block may hold.

`buffer_metric` "chebyshev" (the default) or "euclidean". Chebyshev is exact for square patches: two patches share a pixel when their centres are within the window in both axes.

`seed` Seed for this split only; the training seeds do not move it.

`calibration_frac` Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

Value

A `resample_spec`. resolve_resampling() turns it into folds against the points, and dsm_train() does that itself.

Examples

```
spatial_cv(k = 5, block_size = "auto", buffer = "auto")
# 15% of the points held out to calibrate split conformal intervals
spatial_cv(k = 5, calibration_frac = 0.15)
```

`spatial_folds` *Spatially blocked folds of a point table.*

Description

The fold constructor behind spatial_cv(): the points are binned into square blocks, whole blocks go to one fold, and the buffer then drops the training points too close to a validation or test point.

Usage

```
spatial_folds(
  meta,
  k = 5L,
  test_frac = 0,
  block_size = NULL,
  buffer = NULL,
  buffer_metric = c("chebyshev", "euclidean"),
  test_ids = NULL,
  seed = 42L,
  blocks_per_fold = 10L,
  calibration_frac = 0,
  calibration_ids = NULL
)
```

Arguments

`meta`	Point table: needs x, y and sample_id.
`k`	Number of folds.
`test_frac`	Share held out as the test set, drawn first.
`block_size`	Side of a block, in the units of x/y. NULL sizes it from the extent, for `blocks_per_fold` blocks per fold.
`buffer`	Distance within which a training point is dropped from a fold, in the units of x/y; NULL for none.
`buffer_metric`	"chebyshev" (the default) or "euclidean". Chebyshev is exact for square patches: two patches share a pixel when their centres are within the window in both axes.
`test_ids`	Sample ids that ARE the test set, in place of `test_frac`: a test set drawn again on every run is not a test set.
`seed`	Seed for this split only; the training seeds do not move it.
`blocks_per_fold`	Blocks per fold, for a `block_size` of NULL.
`calibration_frac`	Share of ALL the points held out as a calibration set for the intervals, carved after the test set by the same criterion; 0 for none. It trains nothing and chooses nothing: dsm_final() predicts it and calibrates its "split" interval there.

calibration_ids
: Sample ids that ARE the calibration set, in place of calibration_frac, as test_ids is for the test set.

Value

A fold_plan with k folds.

Examples

```
ex <- example_landscape()
meta <- data.frame(sample_id = seq_len(nrow(ex$profiles)), x = ex$profiles$x,
                   y = ex$profiles$y)
# blocks of 0.05 degrees, and a buffer of the widest window (7 cells of 0.0025)
spatial_folds(meta, k = 3, test_frac = 0.2, block_size = 0.05, buffer = 7 * 0.0025)
```

---

summarise_resamples *Aggregate a unit-level comparison table into one row per config.*

Description

Aggregate a unit-level comparison table into one row per config.

Usage

```
summarise_resamples(
  comparison,
  metrics = c("val_ccc", "val_mae", "val_rmse", "val_r2", "val_mqi", "test_ccc",
    "test_mae"),
  by = "config_id"
)
```

Arguments

comparison	Unit-level table from run_cnn_tuning()/run_cnn_resample().
metrics	Metric column names to aggregate (any that exist).
by	Grouping column; config_id unless you have a reason.

Value

One row per config: n, mean, sd and se of each metric, ranked by the first metric. Failed units are excluded from the statistics but counted in n_failed, because a config that crashes 2 runs in 3 is not the same as one that completed all three.

Examples

```
# A tuning table: three configurations, each on three folds under two seeds
set.seed(1)
cfgs <- data.frame(config_id = c("cfg_001", "cfg_002", "cfg_003"),
                   skill = c(0.615, 0.62, 0.55), n_params = c(2e5, 2e6, 5e4))
cmp <- merge(expand.grid(config_id = cfgs$config_id, fold = 1:3, seed = 1:2,
                         stringsAsFactors = FALSE), cfgs)
cmp$status  <- "success"
cmp$val_ccc <- cmp$skill + rnorm(nrow(cmp), 0, 0.02)
summarise_resamples(cmp, metrics = "val_ccc")
```

---

target_transform_spec *The forward and inverse functions for a named target transform.*

Description

The forward and inverse functions for a named target transform.

Usage

```
target_transform_spec(name)
```

Arguments

name	"none" or "log1p".

Value

list(name, forward, inverse).

Examples

```
tr <- target_transform_spec("log1p")
tr$forward(10)
tr$inverse(tr$forward(10))
```

Index

ale_effect, 4
ale_effect(), *28*, *49*
aoa_di, 5
aoa_reference, 6
aoa_threshold, 7

base::saveRDS(), *47*
block_bootstrap, 8
block_bootstrap(), *9, 10*
block_bootstrap_by_size, 9
block_bootstrap_by_size(), *63*

calc_metrics, 10
caret_available, 11
caret_spec, 11
ccc, 12
check_fold_plan, 13
compare_importance, 13
compare_importance(), *64*
conformal_calibrate, 14
conformal_calibrate(), *18*, *32*
conformal_cv, 15
conformal_interval, 16
conformal_interval(), *22*
conformal_scale_fit, 17
conformal_scale_fit(), *18*
conformal_scaled_calibrate, 17
conformal_scaled_calibrate(), *17*
conformal_scaled_interval, 19
context_importance, 20
context_importance(), *28*
cv_plus_calibrate, 21
cv_plus_calibrate(), *22*
cv_plus_interval, 22
cv_plus_interval(), *21*
cv_residuals, 22
cv_residuals_for_config, 23

di_reference, 24
dissimilarity_index, 25
dsm_final, 25
dsm_final(), *32*, *38*, *41*, *71, 72*
dsm_importance, 27
dsm_importance(), *5*, *20, 21*, *47*, *62*, *64*, *72*, *76*, *82*
dsm_load, 30
dsm_load(), *28*, *41*, *46*
dsm_predict, 31
dsm_predict(), *29*, *48*, *84*
dsm_prepare, 33
dsm_report_final, 36
dsm_train, 37
dsm_train(), *28*, *41*

equal_area_blocks, 39
equal_area_blocks(), *8*, *10*
example_landscape, 40
example_landscape(), *41*
example_run, 41

fold_leakage_report, 41
freeze_selection, 42

get_model, 43

holdout, 43
holdout_cv, 45

importance_groups, 46
importance_groups(), *4*, *28*
importance_map, 47
importance_map(), *29*, *48*, *65*
importance_points, 48
importance_points(), *29*, *47*
importance_weights, 49
importance_weights(), *32*
inside_aoa, 49

knndm_cv, 50
knndm_folds, 51

latest_run_dir, 53
list_models, 54

make_manual_tune_grid, 54
make_tune_grid, 55
model_spec, 56

occlusion_report, 57
one_se, 59

paired_family_test, 60
permutation_importance, 61
permutation_importance(), *20*, *28*, *49*, *81*
picp_report, 62
plot.block_bootstrap_by_size, 63
plot.dsm_importance, 64
plot.importance_comparison, 64
plot.importance_map, 65
plot.spatial_correlogram, 65
prediction_sample, 66
print_aoa, 66
print_noise_floor, 67
print_one_se, 68

random_cv, 68
random_folds, 69
refit_importance, 71
refit_importance(), *28*, *49*
region_cv, 72
region_folds, 73
register_model, 74
resolve_resampling, 75

sage_importance, 76
sage_importance(), *28*, *49*
score_test_grid, 77
seed_noise_floor, 78
selected_config_id, 79
set_torch_threads, 79
setup_torch_device, 80
shap_importance, 81
shap_importance(), *28, 29*, *47*, *49*, *76*
smear, 82
smear(), *83–85*
smear_map, 83
smearing_check, 84
smearing_check(), *83*
smearing_factor, 85
smearing_factor(), *84, 85*
smearing_from_run, 86
smearing_from_run(), *84, 85*
spatial_correlogram, 87
spatial_correlogram(), *65*
spatial_cv, 88
spatial_cv(), *27*, *38*
spatial_folds, 90
summarise_resamples, 91

target_transform_spec, 92
terra::unwrap(), *47*
terra::wrap(), *47*