Package: weightflow (via r-universe)

October 16, 2026

Title Weighting Probability and Non-Probability Survey Samples with Recipe-Aware Replicate Variances

Version 1.3.1

Description Builds survey analysis weights by declaring the whole weighting process as an ordered recipe of explicit adjustments, estimated in a single call: unknown eligibility, within-cluster selection, two-phase subsampling, nonresponse by weighting classes or response-propensity models, calibration and trimming. Calibration to known totals runs on linear, raking or logit distances, bounded, ridge-penalized, model-assisted or integrative with one weight per cluster; a calibrated trim bounds the weights while keeping the control totals exact. Non-probability samples are weighted against a probability reference sample by pseudo-weighting, mass imputation and doubly robust estimators that calibrate a single weight on the fitted values of several working models and on the population totals of the auxiliaries at once, so one multipurpose weight serves many variables. Variances come from a recipe-aware bootstrap and jackknife that re-apply the entire cascade on each replicate, so every adjustment is inside the variance instead of being treated as fixed; across panel waves the replicates are coordinated by carrying forward the selection multiplicities of the sampling units, so the sample overlap enters the variance of a net change as covariance. Rotating and pure panels add attrition, longitudinal weights, gross flows and composite estimation, a self-contained HTML report documents each step, and the weights bridge to the 'survey' and 'srvyr' packages, which are optional: the package itself runs on base R. The methods, and the simulation evidence behind the variance estimators, are described in Ferreira (2026) <doi:10.1177/18747655261484262>.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports stats, utils, graphics, parallel

Suggests MASS, rpart, ranger, testthat (>= 3.0.0), survey, srvyr, dplyr, tidyr, ggplot2, haven, archive, knitr, rmarkdown, spelling, xgboost, yaml

Roxygen list(markdown = TRUE, roclets = c(``rd''))

Config/roxygen2/version 8.0.0

URL https://github.com/jpferreira33/weightflow, https://jpferreira33.github.io/weightflow/

BugReports https://github.com/jpferreira33/weightflow/issues

Config/testthat/edition 3

LazyData true

VignetteBuilder knitr

Repository https://jpferreira33.r-universe.dev

Date/Publication 2026-10-16 01:50:47 UTC

RemoteUrl https://github.com/jpferreira33/weightflow

RemoteRef HEAD

RemoteSha b895824695e80544d2a044f3c56cddb804827e1f

Contents

as_sae_input 4
as_svydesign 5
boot_flows 6
boot_transition 7
bootstrap_estimate 7
bootstrap_weights 9
change_estimate 11
collect_estimates 13
collect_propensities 13
collect_replicate_weights 14
collect_step_detail 16
collect_weights 17
data_defect 18
design_effect 19
disclosure_risk 19
domain_summary 20
jackknife_estimate 21
jackknife_weights 23
level_estimate 25
nr_sensitivity 26
panel_datasets 27
panel_design 28
panel_estimate 30

panel_merge . . . 32
panel_pr . . . 33
plot.prepped_weighting_spec . . . 34
population . . . 34
prep . . . 35
print.weightflow_boot . . . 36
print.weightflow_jack . . . 37
read_recipe . . . 37
reference_sample . . . 38
report_panel . . . 40
report_weighting . . . 41
sample_one . . . 43
sample_survey . . . 44
step_assert . . . 44
step_attrition . . . 45
step_calibrate . . . 49
step_cre . . . 53
step_cross_sectional . . . 56
step_domain . . . 58
step_drop_ineligible . . . 60
step_model_calibration . . . 61
step_nonresponse . . . 65
step_nr_sensitivity . . . 68
step_panel_overlap . . . 70
step_pseudoweight . . . 71
step_rescale . . . 74
step_round . . . 75
step_select_within . . . 77
step_subsample . . . 78
step_trim . . . 80
step_trim_calibrated . . . 81
step_trim_weights . . . 83
step_unknown_eligibility . . . 85
summary.prepped_weighting_spec . . . 87
transition_matrix . . . 87
two_phase_variance . . . 89
wave_bootstrap . . . 90
wave_carry . . . 92
wave_contrast . . . 92
wave_jackknife . . . 93
wave_step . . . 95
weight_factors . . . 97
weightflow-alerts . . . 97
weightflow-concepts . . . 99
weighting_alerts . . . 101
weighting_spec . . . 102
write_recipe . . . 103
y_model . . . 104

Index **105**

`as_sae_input` *Direct estimates and design SEs per domain, ready for small-area estimation*

Description

Small-area estimation (SAE) area-level models (Fay-Herriot) need, for each domain, the *direct* estimate, its *design-based* variance and the effective sample size. This function computes exactly those from a recipe-aware replicate object, so the design-based ingredients flow into `emdi`, `sae` or `hbsae` without leaving the weightflow variance machinery. It does not fit any SAE model itself.

Usage

```
as_sae_input(
  object,
  variable,
  by,
  type = c("mean", "total"),
  level = 0.95,
  cv_breaks = c(0.165, 0.33)
)
```

Arguments

`object`	a `weightflow_boot` or `weightflow_jack` object (from `bootstrap_weights()` / `jackknife_weights()`).
`variable`	name of the study variable.
`by`	name(s) of the domain column(s); several are crossed.
`type`	`"mean"` (default) or `"total"`.
`level`	confidence level for the interval.
`cv_breaks`	two increasing CV cut-points for the publishability rating (default `c(0.165, 0.33)`, common in official statistics): a CV below the first is `"publishable"`, between the two `"review"`, above the second `"not publishable"`.

Details

The domain standard error is the recipe-aware replicate SE (it re-runs the whole recipe per replicate), so it already reflects nonresponse and calibration, not just the final weights.

Value

A data frame with one row per domain: `domain`, `n` (active units), `n_eff` (Kish effective sample size), `estimate`, `se`, `cv`, `ci_lower`, `ci_upper` and `rating`. Pass `estimate` and `se^2` (the sampling variance) to a Fay-Herriot model.

Note for very small domains: the domain estimates share one bootstrap, and a replicate in which any domain has no active unit is dropped for all domains (a conservative choice). With many tiny domains this wastes replicates; use more `replicates` in the bootstrap, or estimate sparse domains in a separate call.

See Also

`domain_summary()`, `bootstrap_estimate()`, `design_effect()`

Other cascade audit: `collect_propensities()`, `collect_step_detail()`, `collect_weights()`, `domain_summary()`, `weight_factors()`, `weighting_alerts()`

Examples

```
spec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region))))
boot <- bootstrap_weights(spec, replicates = 50, strata = "region",
                          psu = "psu", seed = 1)
as_sae_input(boot, "responded", by = "region")
```

---

as_svydesign *Export weightflow weights to a survey design*

Description

`as_svydesign()` builds a linearization (ultimate-cluster) `survey.design` from a prepped recipe, treating the final weights as fixed constants. `as_svrepdesign()` builds a replicate-weights `svyrep.design` from a `bootstrap_weights()` or `jackknife_weights()` object. Both are the bridge to the `survey` package, and therefore to `svytotal()`, `svymean()`, `svyratio()`, `svyby()`, `svyglm()` and domain estimation generally.

Usage

```
as_svydesign(object, ids, strata = NULL, weight_name = ".weight", ...)

as_svrepdesign(object, ...)
```

Arguments

`object`	for `as_svydesign`, a prepped recipe or a data frame with the weight and design columns; for `as_svrepdesign`, a `weightflow_boot` or `weightflow_jack` object.
`ids`, `strata`	column names of the PSU and the stratum.
`weight_name`	name of the weight column.
`...`	passed to the survey constructor.

Details

Only `as_svrepdesign()` propagates the variability of the weighting adjustments (nonresponse, calibration, ...), because each replicate re-runs the whole recipe. `as_svydesign()` is design-based linearization on the *fixed* final weights: its standard errors reflect the sampling design but treat the adjustments as known without error, so they are usually smaller. Use `as_svrepdesign()` (with `bootstrap_weights()` / `jackknife_weights()`) when the adjustment variability should be included.

Value

A `survey.design` / `svyrep.design` object.

See Also

Other variance estimation: `bootstrap_estimate()`, `bootstrap_weights()`, `collect_replicate_weights()`, `jackknife_estimate()`, `jackknife_weights()`

`boot_flows` *Gross-flow TOTALS with standard errors, plus net flows and margins*

Description

The gross change is about the **number of people** who move between states, i.e. totals. From a longitudinal-weight `bootstrap_weights()` object this returns, all with a bootstrap standard error computed within each replicate: the from x to count matrix (the gross flows, in population totals), the **net** flow matrix `i->j minus j->i`, and the margins – the origin totals (how many started in each state), the destination totals (how many ended in each), the stayers (the diagonal) and the movers (off-diagonal).

Usage

```
boot_flows(boot, from, to, states = NULL)
```

Arguments

`boot`	a `weightflow_boot` from `bootstrap_weights()` on the longitudinal recipe.
`from`, `to`, `states`	as in `transition_matrix()`.

Value

a `weightflow_flows` object with `counts`/`counts_se`, `net`/`net_se`, `origin`/`origin_se`, `dest`/`dest_se`, and `stayers`/`movers` (each with its SE).

See Also

`boot_transition()`, `transition_matrix()`

boot_transition *Transition matrix with per-cell bootstrap standard errors*

Description

Like `transition_matrix()`, but every cell also gets a standard error and confidence interval from the bootstrap replicate weights of a longitudinal-weight `bootstrap_weights()` object, by re-tabulating the flow within each replicate. The recipe is re-run per replicate, so the uncertainty of the longitudinal weighting propagates into the flow SEs.

Usage

```
boot_transition(
  boot,
  from,
  to,
  states = NULL,
  format = c("row", "col", "joint", "counts")
)
```

Arguments

`boot`	a `weightflow_boot` from `bootstrap_weights()` on the longitudinal recipe.
`from`, `to`, `states`, `format`	as in `transition_matrix()`.

Value

a `weightflow_transition_boot` with `estimate`, `se`, `ci_lower`, `ci_upper` matrices.

See Also

`transition_matrix()`, `bootstrap_weights()`

bootstrap_estimate *Bootstrap estimate, standard error and confidence interval*

Description

Applies a statistic to the point weights and to every bootstrap replicate, and returns the estimate with its bootstrap standard error and a normal confidence interval. `boot_total()` and `boot_mean()` are the two shortcuts you will use most: a weighted total and a weighted mean of one column.

Usage

```
bootstrap_estimate(
  boot,
  statistic,
  level = 0.95,
  ci_type = c("normal", "t", "percentile"),
  df = NULL
)

boot_total(
  boot,
  variable,
  level = 0.95,
  ci_type = c("normal", "t", "percentile"),
  df = NULL
)

boot_mean(
  boot,
  variable,
  level = 0.95,
  ci_type = c("normal", "t", "percentile"),
  df = NULL
)
```

Arguments

boot	a weightflow_boot object.
statistic	a function function(w, data) returning a numeric scalar (or vector) given a weight vector and the data.
level	confidence level for the interval.
ci_type	interval type: "normal" (default, z-based), "t" (Student t with the design degrees of freedom, wider and less anticonservative with few PSUs), or "percentile" (empirical quantiles of the valid replicates).
df	degrees of freedom for the t interval; NULL (default) uses the design df stored on the object (total PSUs minus strata).
variable	name of the variable to estimate.

Details

The bootstrap variance takes the replicate estimates $\hat{\theta}_b^*$ around the point estimate $\hat{\theta}$ (the mse = TRUE convention of survey), over the R valid replicates (a failed replicate is dropped, not counted),

$$\widehat{V}_{\mathrm{boot}}(\hat{\theta}) = \frac{1}{R}\sum_{b=1}^{R}\left(\hat{\theta}_b^* - \hat{\theta}\right)^2.$$

Value

A data frame with `estimate`, `se`, `ci_lower`, `ci_upper`.

See Also

Other variance estimation: `as_svydesign()`, `bootstrap_weights()`, `collect_replicate_weights()`, `jackknife_estimate()`, `jackknife_weights()`

Examples

```
spec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region))))
boot <- bootstrap_weights(spec, replicates = 50, strata = "region",
                          psu = "psu", seed = 1)
# a t interval with the design degrees of freedom (safer with few PSUs)
bootstrap_estimate(boot, function(w, d) sum(w * d$responded), ci_type = "t")
```

`bootstrap_weights` *Recipe-aware bootstrap replicate weights*

Description

Builds bootstrap replicate weights by resampling primary sampling units (PSUs) with replacement within strata and re-running the **entire** weighting recipe on each replicate – every estimated stage (nonresponse, calibration, model calibration, trimming), not just one. Reach for this when those stages are estimated from the sample and you want their uncertainty inside the standard error, instead of conditioning on them as if they were known.

Usage

```
bootstrap_weights(
  object,
  replicates = 200L,
  strata = NULL,
  psu = NULL,
  m = NULL,
  fpc = NULL,
  lonely_psu = c("certainty", "collapse"),
  seed = NULL,
  cores = 1L,
  progress = TRUE,
  .tp_component = c("full", "phase1", "phase2")
)
```

Arguments

`object`	a `weighting_spec` (or a prepped one) holding the recipe.
`replicates`	number of bootstrap replicates.
`strata, psu`	column names of the stratum and the PSU. If `psu` is NULL each unit is its own PSU; if `strata` is NULL a single stratum is assumed.
`m`	PSUs drawn per stratum (default `n - 1`).
`fpc`	optional first-stage finite-population correction: the name of a column holding the first-stage sampling fraction f_h (constant within stratum, in `[0, 1]`), a single number applied to every stratum, or a numeric vector named by stratum level. `NULL` (default) is the with-replacement bootstrap (no correction). The correction folds `(1 - f_h)` into the Rao-Wu rescaling (Rao, Wu and Yue 1992; Beaumont and Patak 2012); `f_h = 0` reproduces the uncorrected result. Only available for the bootstrap.
`lonely_psu`	how to treat strata with a single PSU (which a with-replacement bootstrap cannot resample): "certainty" (default) treats them as self-representing, so they contribute no bootstrap variance, and warns; "collapse" merges the single-PSU strata into a pseudo-stratum (with the smallest other stratum if there is only one), so they are resampled and do contribute a (conservative) variance. For full control, build your own collapsed stratum column and pass it as `strata`.
`seed`	optional RNG seed.
`cores`	number of parallel workers for the replicates (default 1 = serial). With `cores > 1` the replicate re-preps run in parallel via `parallel::mclapply` (forking; on Windows it falls back to serial). Results are identical to the serial run: the resampling is drawn up front with the seed and only the deterministic re-prep is parallelised.
`progress`	print progress every 25 replicates (serial only).
`.tp_component`	internal. For a two-phase recipe, `"full"` (default) draws the coupled factor; `"phase1"` / `"phase2"` draw only the phase-1 or phase-2 component of the coupling. Used by `two_phase_variance()` to split $V = V_1 + V_2$; not for direct use.

Details

The multiplier is the Rao-Wu rescaling bootstrap: within a stratum with n PSUs, m PSUs are drawn with replacement (default $m = n - 1$) and unit i in PSU k gets $\lambda = 1 - \sqrt{m/(n-1)} + \sqrt{m/(n-1)}\,(n/m)\,t_k$, with t_k the number of times its PSU was drawn.

Value

An object of class `weightflow_boot` with the `replicates` matrix (units x replicates), the point `weights`, and the design metadata.

See Also

Other variance estimation: `as_svydesign()`, `bootstrap_estimate()`, `collect_replicate_weights()`, `jackknife_estimate()`, `jackknife_weights()`

Examples

```
spec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region))))
boot <- bootstrap_weights(spec, replicates = 50, strata = "region",
                          psu = "psu", seed = 1)
boot_total(boot, "responded")
# with a first-stage finite-population correction (per-stratum sampling fraction)
d <- sample_survey; d$f <- 0.1
spec_f <- weighting_spec(d, base_weights = pw)
bootstrap_weights(spec_f, replicates = 50, strata = "region", psu = "psu",
                  fpc = "f", seed = 1)
```

---

change_estimate *Net change between two panel waves, with honest variance*

Description

Estimates a net change (a statistic in wave 2 minus the same statistic in wave 1) from a `wave_bootstrap()`, decomposing its variance into `V1`, `V2` and the sampling covariance the overlap induces: `V = V1 + V2 - 2*Cov`. Because the bootstrap replicates are coordinated by PSU, `Cov` emerges from them – no analytic covariance formula. Reports the correlation `rho` and `deff_change = V / (V1 + V2)`, i.e. how much the overlap lowered the variance of the change relative to treating the waves as independent.

Usage

```
change_estimate(
  wb,
  statistic,
  waves = NULL,
  level = 0.95,
  type = c("absolute", "relative"),
  by = NULL,
  ci_type = c("normal", "t", "percentile"),
  df = NULL
)

change_mean(
  wb,
  variable,
  waves = NULL,
  level = 0.95,
  ci_type = c("normal", "t", "percentile"),
  df = NULL,
  type = c("absolute", "relative"),
  by = NULL
)
```

```
change_total(
  wb,
  variable,
  waves = NULL,
  level = 0.95,
  ci_type = c("normal", "t", "percentile"),
  df = NULL,
  type = c("absolute", "relative"),
  by = NULL
)
```

Arguments

wb	a weightflow_wave_boot from wave_bootstrap().
statistic	a function function(w, data) returning a single number, evaluated on each wave's weights and data (e.g. function(w, d) weighted.mean(d$y, w)).
waves	optional length-2 character vector naming the two waves (defaults to the first two).
level	confidence level for the interval.
type	"absolute" (default) for the net change theta2 - theta1, or "relative" for the relative change theta2 / theta1 - 1 (e.g. the percent change of a rate). The relative change's variance comes from the coordinated replicate distribution of the ratio (bootstrap) or the delta method on the level pieces (jackknife), so the overlap covariance is captured either way. V1, V2, cov, rho always describe the levels.
by	optional name of a grouping column present in both waves: returns one change per domain (a weightflow_change_by table) instead of a single change.
ci_type	interval type: "normal" (default, z-based), "t" (Student t with df degrees of freedom, wider and safer with few PSUs) or "percentile" (bootstrap only).
df	degrees of freedom for the "t" interval; NULL (default) uses the design df stored on the object (total PSUs minus strata).
variable	name of a numeric variable present in both waves.

Value

an object of class weightflow_change with the estimate, se, V, V1, V2, Vind = V1 + V2, cov, rho, deff_change and the confidence interval; or, with by=, a weightflow_change_by holding one row per domain.

See Also

wave_bootstrap(), change_mean(), change_total(), level_estimate()

collect_estimates *Evaluate an estimation pipeline*

Description

Runs every estimand of a `weightflow_estimation` across the cross of its `step_domain()` columns, using the coordinated replicates for the variance, and returns a tidy table.

Usage

```
collect_estimates(est)
```

Arguments

`est`	a `weightflow_estimation` from `step_estimate()`.

Value

a `weightflow_estimation_result`: one row per (estimand x domain cell) with `estimate`, `se`, `ci_lower`, `ci_upper` and (for changes) `rho`. The object also carries a row-aligned `detail` frame (confidence level, effective replicates, the two wave levels behind a change and the overlap design effect) and the pipeline's `filters`, so `report_panel()` can document the estimates without re-running them.

See Also

`step_estimate()`, `report_panel()`

collect_propensities *Recover the fitted response propensities of a nonresponse step*

Description

A `step_nonresponse(method = "propensity")` step fits a response-propensity model and adjusts the weights by $1/\hat{p}$. `prep()` keeps the full per-unit propensity vector $\hat{p}$ (out-of-fold when cross-fitting is used) on the step, but it is not returned by `collect_weights()`. This accessor extracts it aligned to the sample, so you can inspect its distribution and confirm the nonresponse model is well fitted before trusting the adjusted weights. It works the same way whether the adjustment was made at the unit level or, through `cluster`, at the household level (there the household propensity is broadcast to its members).

Usage

```
collect_propensities(object, step = NULL)
```

Arguments

object	a prepped weighting_spec (the output of prep()).
step	optional integer, which step to read when the recipe has more than one propensity step. If NULL (default) and there is a single propensity step it is used; with several, the last one is used with a message.

Value

The sample data.frame with columns appended: .propensity (the fitted response propensity $\hat{p}$, NA for units outside the model, i.e. ineligible / already dropped), .responded (the response indicator the model used), .weight_in (the weight reaching the step, see below), .factor (the multiplier the step actually applied to the unit), .status (a factor that labels each unit as "eligible respondent", "eligible nonrespondent" or "not in propensity model"), and, when the step uses propensity classes (num_classes), .class (the assigned class). Units not in the propensity model carry NA in the per-unit columns. .weight_in is the weight *reaching* the nonresponse step – it already carries any earlier adjustment (unknown-eligibility redistribution, within-cluster selection), not the raw base weight. At the unit level it is also the weight the propensity model is fitted with (unless weight_model = FALSE); with cluster, the model is fitted at the household level with the household weight, which equals .weight_in only when weights are uniform within the household. .factor equals $1/\hat{p}$ only when num_classes = NULL; with propensity classes it is the class-level adjustment, so 1/.propensity does not reconstruct the applied factor – use .factor. The stage-by-stage weights are available through weight_factors() and domain_summary().

See Also

step_nonresponse(), collect_weights()

Other cascade audit: as_sae_input(), collect_step_detail(), collect_weights(), domain_summary(), weight_factors(), weighting_alerts()

Examples

```
fit <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "propensity",
                   formula = ~ sex + region, engine = "logit") |>
  prep()
p <- collect_propensities(fit)
summary(p$.propensity)
```

---

collect_replicate_weights

Collect replicate weights into a data frame ready for srvyr

Description

Returns the data with the point weight and every replicate weight as ordinary columns, plus the replication design as attributes. This is the form srvyr::as_survey_rep() and survey::svrepdesign() expect, and the form to write out when the analysis continues in another session, another script or another language.

Usage

```
collect_replicate_weights(
  object,
  weight_name = ".weight",
  prefix = "rep_",
  drop_zero = TRUE,
  scramble = FALSE
)
```

Arguments

object	a weightflow_boot or weightflow_jack object.
weight_name	name of the point-weight column to add.
prefix	prefix for the replicate-weight columns (rep_1, rep_2, ...).
drop_zero	keep only active units (point weight > 0).
scramble	disclosure control for a public-use file. When TRUE, the replicate columns are randomly permuted (their rscales move with them, so the variance is unchanged) and the design identifier columns (the strata and psu columns used to build the replicates) are dropped from the output, so the exported weights do not reveal the sampling design. The point weights and the variance estimate are unaffected. Set a seed beforehand for a reproducible permutation. The result carries attribute "scrambled" = TRUE.

Value

A data frame: the original columns, weight_name, and one column per replicate. The number of replicates is in attribute "R", and the replication design in attributes "type", "scale" and "rscales".

See Also

Other variance estimation: as_svydesign(), bootstrap_estimate(), bootstrap_weights(), jackknife_estimate(), jackknife_weights()

Examples

```
spec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region))))
boot <- bootstrap_weights(spec, replicates = 30, strata = "region",
                          psu = "psu", seed = 1, progress = FALSE)
df <- collect_replicate_weights(boot)   # or a weightflow_jack object
if (requireNamespace("srvyr", quietly = TRUE) &&
    requireNamespace("dplyr", quietly = TRUE)) {
  srvyr::as_survey_rep(df, weights = .weight,
                       repweights = dplyr::starts_with("rep_"),
                       type = attr(df, "type"), combined.weights = TRUE,
                       scale = attr(df, "scale"), rscales = attr(df, "rscales"),
                       mse = TRUE)
```

```
}
```

`collect_step_detail` *Per-unit detail of one step of the cascade*

Description

A generic companion to `collect_weights()` that returns, for a single step, the weight it received and the multiplier it applied to every unit, plus any quantities that step computed internally (for a propensity step, the fitted propensity and its class). `.weight_in` and `.factor` are read from the stage-by-stage weights that `prep()` already stores, so `.weight_in * .factor` equals the weight leaving the step by construction, for any step. Native columns (those a step exposes on its own) are `NA` for units the step did not touch.

Usage

```
collect_step_detail(object, step = NULL)
```

Arguments

`object`	a prepped `weighting_spec` (the output of `prep()`).
`step`	optional: which step to inspect, as an integer position (1 for the first piped step) or a step id string (e.g. "calibrate_1"; see the recipe print-out). If `NULL` (default): a single step exposing native detail is used; if several do, or if none do and the recipe has more than one step, an error lists the steps so you can choose.

Value

The sample `data.frame` with `.weight_in` (the weight reaching the step, carrying every earlier adjustment) and `.factor` (the multiplier the step applied to each unit, `NA` where the incoming weight is zero) appended, plus any native columns of the chosen step (for a propensity step: `.propensity`, `.responded`, and `.class` when propensity classes are used), which are `NA` outside the units the step covers. Here `.factor` is defined for every unit with a nonzero incoming weight, so an active unit the step did not touch reports `.factor = 1`; this differs from `collect_propensities()`, where `.factor` is `NA` outside the propensity model (see its `.status`).

See Also

`collect_weights()`, `collect_propensities()`, `weight_factors()`

Other cascade audit: `as_sae_input()`, `collect_propensities()`, `collect_weights()`, `domain_summary()`, `weight_factors()`, `weighting_alerts()`

Examples

```
fit <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "propensity",
                   formula = ~ sex + region, engine = "logit") |>
  prep()
d <- collect_step_detail(fit, step = 1)
head(d[!is.na(d$.factor), c(".weight_in", ".factor", ".propensity")])
```

`collect_weights` *Extract the data with the computed weights*

Description

Returns the sample as a `data.frame` with the final analysis weight attached as a column, ready to hand to an estimation routine. By default the units that left the cascade (weight 0) are dropped, so what comes back is the responding, in-scope sample and its weights.

Usage

```
collect_weights(
  object,
  drop_zero = TRUE,
  keep_intermediate = FALSE,
  weight_name = ".weight"
)
```

Arguments

`object`	a prepped object (output of prep()).
`drop_zero`	logical. If TRUE, drops rows with final weight 0 (ineligible / nonresponse). Default TRUE.
`keep_intermediate`	logical. If TRUE, adds one column per stage.
`weight_name`	name of the final weight column. Default ".weight".

Value

data.frame.

See Also

Other cascade audit: `as_sae_input()`, `collect_propensities()`, `collect_step_detail()`, `domain_summary()`, `weight_factors()`, `weighting_alerts()`

Examples

```
fitted <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  prep()
head(collect_weights(fitted))
```

`data_defect` *Data-defect diagnostics for a non-probability sample*

Description

For a non-probability sample, Meng (2018) decomposes the error of the sample mean into the data-defect correlation (`rho`, the population correlation between the target variable and the participation indicator), the sampling fraction and the problem difficulty. The effective sample size a probability sample would need to match the same mean-squared error is

$$n_{\text{eff}} = \frac{f/(1-f)}{\rho^2}, \qquad f = n/N,$$

which does not depend on the outcome except through `rho`. Because `rho` on the target variable is not observable from the sample alone, `data_defect()` returns the effective size across a grid of plausible residual `rho` (read it as an ignorance range, not a single number), plus the measurable selection strength on the covariates used for pseudo-weighting (the largest correlation between an auxiliary and participation, which pseudo-weighting neutralises).

Usage

```
data_defect(object, ddc_grid = c(0.001, 0.005, 0.01, 0.05, 0.1))
```

Arguments

`object`	a prepped non-probability `weighting_spec` (from `prep()`, built with `weighting_spec(..., nonprob = TRUE)`).
`ddc_grid`	the residual data-defect correlations to tabulate. Positive values; only their magnitude matters.

Value

a list (class `weightflow_data_defect`) with the sample size `n`, the estimated population size `N` (the sum of the final weights), the fraction `f`, the sensitivity `grid` (`ddc`, `n_eff`), and `aux`, a data frame of the covariate-participation correlations (or `NULL` when the recipe has no pseudo-weighting step).

References

Meng, X.-L. (2018). Statistical paradises and paradoxes in big data (I). Annals of Applied Statistics 12(2), 685-726.

See Also

`step_pseudoweight()`, `report_weighting()`

design_effect *Kish design effect from unequal weighting*

Description

Computes Kish's design effect due to unequal weighting, $deff = 1+CV^2(w) = m\sum w^2/(\sum w)^2$, and the effective sample size $n_{\text{eff}} = m/deff$ it implies. It is the standard one-number summary of what a weighting cascade cost in precision, and it is what the `summary()` and `plot()` methods of a prepped recipe report step by step.

Usage

```
design_effect(w)
```

Arguments

`w`	vector of weights (zeros are dropped; negative weights are kept active, but see the note above on the design effect).

Details

Zero weights are dropped (they are the "dropped-unit" marker); negative weights – a valid but unusual output of unbounded linear/GREG calibration – are kept active, so the count `n` matches `collect_weights()`. Be aware, however, that the Kish formula assumes non-negative weights: a negative weight shrinks $\sum w$ and enlarges $\sum w^2$ at once, so with negatives present `deff` is inflated and no longer interpretable as an effective-sample summary. `prep()` raises an alert when a calibration produces negative weights; prefer `bounds` to keep the factor positive if you need the design effect to be meaningful.

Value

list with deff, n_eff, cv and n.

Examples

```
design_effect(sample_survey$pw)
```

disclosure_risk *Flag re-identification risk from outlier weights within a publication cell*

Description

A unit whose final weight is far larger than the rest of its publication cell is a disclosure risk in a public-use file: an extreme weight makes a rare unit stand out. `disclosure_risk()` flags, within each cell defined by `by`, the units whose final weight exceeds `ratio` times the cell's median weight, and reports the unit's share of the cell's total weight. Trimming (`step_trim_weights()` or the totals-preserving `step_trim_calibrated()`) is the usual remedy.

Usage

```
disclosure_risk(object, by, ratio = 10)
```

Arguments

object	a prepped weighting_spec (the output of prep()).
by	the name(s) of the publication cell column(s) (e.g. "region", or c("region", "sex")). The risk is judged within each cell.
ratio	the multiple of the cell median weight above which a unit is flagged. Default 10.

Value

A data.frame, one row per flagged unit, with the row index (.row), the cell, the unit weight, the cell_median, the cell_n (active units in the cell) and cell_share (the unit's fraction of the cell's total weight), ordered from the largest weight down. Zero rows when nothing is flagged.

See Also

step_trim_weights(), step_trim_calibrated(), collect_replicate_weights()

Examples

```
fit <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  prep()
disclosure_risk(fit, by = "region")
```

---

domain_summary *Per-domain weight summary at every stage of the cascade*

Description

For quality control by study domain (for example a department / DAM), this summarises how the weights move within each domain at every stage of the recipe: the base weights, then the weights after each step. It reads the stage-by-stage weights that prep() already stores, so it adds no computation to the cascade and never changes a weight.

Usage

```
domain_summary(object, by, min_n_eff = NULL)
```

Arguments

object	a prepped weighting_spec (the output of prep()).
by	the name(s) of one or more domain columns in the data (e.g. "region", or c("region", "area") to cross them). Domains are ordered by the factor levels of the column (or numerically for a numeric column); units with a missing domain value are shown as a "(missing)" domain rather than dropped silently.
min_n_eff	optional publication threshold. When set to a positive number, the result gains a logical publishable column (whether the domain's final-stage effective sample size reaches the threshold) and a warning names the domains that fall below it, turning the implicit reliability read into an explicit gate. Domains below the threshold are candidates for small-area estimation (see as_sae_input()) rather than direct estimation.

Value

A data.frame with one row per stage x domain and the columns stage (an ordered factor: base weights, then 1. <step>, 2. <step>, ...), domain (an ordered factor), n_active (active units in the domain at that stage), sum_w (sum of the active weights), mean_w, deff (the Kish design effect within the domain) and n_eff; and, when min_n_eff is given, publishable. Reading down a domain shows how its weight total and dispersion evolve step by step.

See Also

design_effect(), weight_factors(), summary.prepped_weighting_spec()

Other cascade audit: as_sae_input(), collect_propensities(), collect_step_detail(), collect_weights(), weight_factors(), weighting_alerts()

Examples

```
fit <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
 step_calibrate(method = "raking", margins = list(region = c(table(population$region)))) |>
  prep()
domain_summary(fit, by = "region")
```

---

jackknife_estimate *Jackknife estimate, standard error and confidence interval*

Description

Applies a statistic to the point weights and to every delete-a-PSU replicate, and returns the estimate with its stratified jackknife (JKn) standard error and a normal confidence interval. jack_total() and jack_mean() are the shortcuts for a weighted total and a weighted mean of one column.

Usage

```
jackknife_estimate(
  jack,
  statistic,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)

jack_total(jack, variable, level = 0.95, ci_type = c("normal", "t"), df = NULL)

jack_mean(jack, variable, level = 0.95, ci_type = c("normal", "t"), df = NULL)
```

Arguments

jack	a weightflow_jack object.
statistic	a function function(w, data) returning a numeric scalar (or vector) given a weight vector and the data.
level	confidence level for the interval.
ci_type	interval type: "normal" (default) or "t" (Student t with the design degrees of freedom). The percentile interval is not defined for the jackknife.
df	degrees of freedom for the t interval; NULL (default) uses the design df stored on the object (total PSUs minus strata).
variable	name of the variable to estimate (for jack_total/jack_mean).

Details

The stratified (JKn) variance sums each stratum's delete-a-PSU spread,

$$\widehat{V}_{JK} = \sum_{h} \frac{n_h - 1}{n_h} \sum_{i \in h} \left(\hat{\theta}_{(hi)} - \hat{\theta}_h\right)^2,$$

with $\hat{\theta}_{(hi)}$ the estimate with PSU i of stratum h deleted and $\hat{\theta}_h$ their within-stratum mean; the unstratified JK1 uses a single stratum. No finite population correction is applied.

Value

A data frame with estimate, se, ci_lower, ci_upper.

Note

jack_total() / jack_mean() center the replicate deviations on the per-stratum mean of the deleted-PSU estimates (the standard JKn). The survey design built by as_svrepdesign() instead uses mse = TRUE, which centers on the point estimate. Both are legitimate, so the standard errors from jack_total() and from svytotal() on the same object can differ slightly.

See Also

Other variance estimation: as_svydesign(), bootstrap_estimate(), bootstrap_weights(), collect_replicate_weights(), jackknife_weights()

Examples

```
spec <- weighting_spec(sample_one, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region))))
jk <- jackknife_weights(spec, strata = "region", psu = "psu", progress = FALSE)
jackknife_estimate(jk, function(w, d) sum(w * d$employed, na.rm = TRUE))
```

---

jackknife_weights *Recipe-aware delete-a-PSU jackknife replicate weights*

Description

Builds jackknife replicate weights by deleting one primary sampling unit (PSU) at a time and re-running the **entire** weighting recipe on each replicate. This is the deterministic sibling of bootstrap_weights(): same recipe-aware variance, no random number generation, and a replicate count fixed by the design rather than chosen by the analyst.

Usage

```
jackknife_weights(
  object,
  strata = NULL,
  psu = NULL,
  groups = NULL,
  seed = NULL,
  lonely_psu = c("certainty", "collapse"),
  max_replicates = 5000L,
  cores = 1L,
  progress = TRUE
)
```

Arguments

object	a weighting_spec (inert recipe) or a prepped weighting_spec. Pass the recipe *before* prep(): the jackknife preps it once per replicate.
strata	name of the stratum column, or NULL for a single stratum.
psu	name of the PSU column, or NULL to delete one unit at a time.
groups	NULL (default) or the number of random groups G for a delete-a-group jackknife on a direct sample. Units are assigned to groups at random within each stratum and one group is deleted per replicate, giving G replicates instead of one per unit. Requires psu = NULL: the group replaces the PSU, it does not nest inside one.

seed	optional integer seed for the random group assignment, so the replicates are reproducible. The caller's RNG state is restored on exit.
lonely_psu	how to treat strata with a single PSU: "certainty" (default) skips them (no variance) and warns; "collapse" merges them into a pseudo-stratum so they yield delete-a-PSU replicates.
max_replicates	the largest number of replicates this call will build (default 5000); above it the call stops rather than starting the run. Unlike bootstrap_weights(), where you set replicates, here the count falls out of the design: one replicate per PSU, and with psu = NULL one per **unit**. Each replicate re-runs the whole recipe and holds a column of weights, so a direct sample of 100,000 units means 100,000 re-preps and an 80 GB matrix. That is the correct delete-one jackknife, which is why the limit refuses it up front instead of letting the session die: for a sample that size the usual device is the delete-a-group jackknife, groups = 50. Set Inf to build them all.
cores	number of parallel workers for the replicates (default 1 = serial). With cores > 1 the replicate re-preps run in parallel via parallel::mclapply (forking; serial on Windows). For a deterministic recipe the result is identical to the serial run.
progress	print progress every 25 replicates (serial only).

Details

For a stratum h with n_h PSUs, the replicate that deletes PSU i zeros the base weight of that PSU and inflates the remaining PSUs of the stratum by $n_h/(n_h - 1)$; other strata are unchanged. There is one replicate per PSU. Strata with a single PSU contribute no variance and are skipped. This is the stratified jackknife (JKn); with strata = NULL it is the unstratified jackknife (JK1), and with psu = NULL each unit is its own PSU (delete-one-unit jackknife).

With a direct (element) sample the delete-one-unit jackknife is the right estimator, but it needs one replicate per unit: the whole recipe is re-prepped n times and the replicate matrix is n x n, which is 80 GB at n = 100,000. groups gives the standard device for that case, the **delete-a-group** jackknife (Kott 2001; Rust and Rao 1996): the sample is partitioned at random into G groups within each stratum and one whole group is deleted at a time, so there are G replicates instead of n. It is the same estimator the engine already computes – the group simply plays the part of the PSU – with $(G - 1)/G$ in place of $(n_h - 1)/n_h$, and df = G - strata, which is the honest precision of the variance. G between 30 and 100 is the usual range; it is not defaulted, because the choice is the analyst's and the delete-one and delete-a-group variances are not the same number (the second is noisier).

Value

An object of class weightflow_jack with the replicates matrix (units x replicates), the point weights, the per-replicate stratum and stratum size (used by jackknife_estimate()), and the design metadata.

See Also

Other variance estimation: as_svydesign(), bootstrap_estimate(), bootstrap_weights(), collect_replicate_weights(), jackknife_estimate()

Examples

```
spec <- weighting_spec(sample_one, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region))))
jk <- jackknife_weights(spec, strata = "region", psu = "psu", progress = FALSE)
jack_total(jk, "employed")
```

`level_estimate` *Level estimate for a single panel wave, with its replicate variance*

Description

Estimates a statistic in ONE wave of a `wave_bootstrap()` or `wave_jackknife()` object, with the variance from that wave's replicates. Lets you report a level and a change from the same coordinated object.

Usage

```
level_estimate(
  wb,
  statistic,
  wave = NULL,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)

level_mean(
  wb,
  variable,
  wave = NULL,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)

level_total(
  wb,
  variable,
  wave = NULL,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)
```

Arguments

wb	a weightflow_wave_boot or weightflow_wave_jack.
statistic	a function function(w, data) returning one number.
wave	optional wave label (defaults to the first).
level	confidence level for the normal interval.
ci_type	"normal" (default, z) or "t" (Student t with df df).
df	degrees of freedom for the "t" interval; NULL uses the object's design df.
variable	name of a numeric variable in the wave's data.

Value

an object of class weightflow_level with estimate, se, V and the interval.

See Also

change_estimate(), level_mean(), level_total()

---

nr_sensitivity *Read the nonresponse-sensitivity analysis from a prepped recipe*

Description

Returns the proxy pattern-mixture ignorance analysis stored by step_nr_sensitivity(): the adjusted mean of the study variable at each phi, with the ignorance interval and the proxy strength.

Usage

```
nr_sensitivity(object, step = NULL)
```

Arguments

object	a prepped weighting_spec containing a step_nr_sensitivity().
step	optional step id to select among several sensitivity steps (for example one per study variable). With one step it can be left NULL.

Value

a list (class weightflow_nr_sensitivity) with table (phi, mu), rho, ybar_r, ignorance (the min-max interval over phi), mu_mar (the phi = 0 estimate) and the respondent/nonrespondent counts.

See Also

step_nr_sensitivity()

panel_datasets *Synthetic rotating- and pure-panel datasets*

Description

Four small, reproducible household-panel datasets that ship with weightflow to test and illustrate the panel tools. They share one structure and differ only in the **rotation system**, so the same code runs on a pure panel and on each rotating design. All are in **long format**: one row per person and per wave the person is in sample. Continuing units keep the same `household_id` / `person_id` across waves, so they link the panel; the household is the natural cluster and `c(household_id, person_no)` the person-level key.

Usage

```
panel_puro

panel_cl

panel_ine

panel_us
```

Format

A `data.frame` with one row per person-wave and the columns:

household_id household id, persistent across waves (the panel link / cluster).

person_id, person_no person id and person-number within household; `c(household_id, person_no)` is the person key.

stratum, psu design stratum and primary sampling unit (PSU nested in stratum, at least two PSUs per stratum), for the coordinated bootstrap / jackknife.

region, sex, age covariates usable as estimation domains.

wave wave (month) index.

rotation_group, month_in_sample rotation group and order-in-sample.

pw design (base) weight.

disposition between-wave disposition: `"R"`, `"NR"`, `"OS"`, `"UNK"`.

lf_status labour status within the working-age population, a factor with levels `"emp"` / `"unemp"` / `"inact"`; `NA` for non-respondents. This is the `status` argument of `step_cre()` (the previous-wave composite auxiliary).

employed, unemployed employed / unemployed indicators (labour force; `NA` if not `"R"` or not in the labour force).

income labour income (`NA` if not `"R"`).

Details

The between-wave disposition `disposition` follows the four-state taxonomy the longitudinal cascade needs: `"R"` responded, `"NR"` eligible nonresponse (reweight), `"OS"` out of scope – left the target population between waves, so the household exits permanently and is not reweighted – and `"UNK"` unknown eligibility. The variables of interest (`employed`, `unemployed`, `income`) are observed only when `disposition == "R"` (and, for the labour-force items, when the person is in the labour force), otherwise `NA`; they repeat across waves for continuing persons, with within-person persistence, so net change and gross flows are meaningful.

The datasets differ only in the rotation calendar (which waves each rotation group is in sample), giving different overlap structures:

`panel_puro` **Pure panel**, no rotation: every unit is followed across all 4 waves; the sample shrinks only through attrition. Style of EU-SILC / SLID pure panels.

`panel_cl` **Chile ENE, 2-2-2** (in-out-in): 3 waves, consecutive overlap ~1/2, and units that **return** (in sample in waves 1 and 3 but not 2). The real ENE has no public rotation group; `rotation_group` is included for teaching, but the panel also links through the persistent ids alone.

`panel_ine` **INE Uruguay ECH / StatCan LFS, 6-month rotation**: 6 groups in sample each wave, one sixth rotating out per wave, so consecutive overlap ~5/6.

`panel_us` **US CPS, 4-8-4**: 3 waves, consecutive overlap ~3/4, with a cohort that leaves and **returns** after the 8-month gap.

Examples

```
# rotation structure of the 6-month panel
panel_design(panel_ine, unit = c("household_id", "person_no"), wave = "wave",
             rotation_group = "rotation_group", pattern = "6")
# coordinated change of the unemployment rate between two waves
t1 <- subset(panel_ine, wave == 1 & disposition == "R")
t2 <- subset(panel_ine, wave == 2 & disposition == "R")
wb <- wave_bootstrap(
  list(T1 = weighting_spec(t1, base_weights = pw),
       T2 = weighting_spec(t2, base_weights = pw)),
  replicates = 100, strata = "stratum", psu = "psu", seed = 1, progress = FALSE)
change_mean(wb, "unemployed")
```

`panel_design` *Describe the rotating-panel structure of a survey*

Description

Tags stacked per-wave survey data with its rotation structure so the panel steps and the report can read it, and so the overlap and linkage quality can be inspected *before* any weighting. `panel_design()` computes only; it does not create or change weights. It is the panel analogue of `reference_sample()`: the descriptor lives in `attr(data, "wf_panel")` and the returned object is still an ordinary `data.frame`.

Usage

```
panel_design(
  data,
  unit,
  wave,
  rotation_group = NULL,
  cluster = NULL,
  pattern = NULL,
  waves = NULL,
  reference_wave = NULL
)
```

Arguments

data	a stacked data.frame, one row per unit per wave.
unit	one or more column names (a character vector) that together identify the longitudinal unit, stable across waves. A household is often a single id ("ID"); a person needs several (c("ID", "person_no") = household id plus person line number). The columns are pasted into the tracking key.
wave	string: the column holding the wave/period.
rotation_group	optional string: the column holding the rotation group / panel. Needed to derive Pr(panel selection) exactly; without it that field is left NA and only the unit-level overlap is computed.
cluster	optional one or more column names identifying a within-unit cluster (e.g. the household "ID") when the tracked unit is a person but the overlap is realised at the household level. Given one, the descriptor also carries overlap_cluster – the same wave x wave overlap matrix computed over distinct clusters – plus n_clusters and n_linked_clusters. In a rotating household panel the dwelling stays in sample while its members change, so the cluster overlap is the one the rotation calendar describes and the unit-level one is that figure net of within-household churn; the gap between them is the churn.
pattern	optional string describing the rotation scheme, for verification only: CEPAL's "4(0)1", the CPS "4-8-4", or a plain integer like "6". Nothing in the computation depends on parsing it.
waves	optional character vector giving the wave order explicitly; defaults to sort(unique(data[[wave]])).
reference_wave	optional wave whose population the longitudinal weight represents (the calibration target for a longitudinal recipe); defaults to the first wave. Read by step_longitudinal().

Details

data must be in **stacked (long) form**: one row per unit per wave, with a unit column that is stable across waves and a wave column identifying the period. From the unit x wave membership it derives the observed overlap matrix (the fraction of each wave retained in every other wave, i.e. CEPAL's *traslape*), and, when rotation_group is given, the panel-selection probability Pr(panel selection) for each adjacent pair and for the full combination – the reciprocal of which is the CEPAL panel base-weight factor.

The linkage quality is measured, not assumed: if a `pattern` is supplied the observed adjacent overlap is compared against the one it implies, and a large gap (alert `PN-01`) is the early warning that the linkage key is unstable (relabelled ids, a redesigned frame, duplicated panels). Uneven rotation-group sizes, which break the scalar reciprocal of `Pr(panel selection)`, raise `PN-02`.

Value

`data`, unchanged as a data frame, with the descriptor in `attr(data, "wf_panel")` and class `"wf_panel_design"` prepended.

See Also

`panel_merge()`, `reference_sample()`

Examples

```
# person-level tracking: the key is household id + person line number
pd <- panel_design(panel_ine, unit = c("household_id", "person_no"), wave = "wave",
                   rotation_group = "rotation_group", cluster = "household_id",
                   pattern = "6")
pd            # overlap matrix, Pr(panel selection), linkage rate, alerts
summary(pd)
```

`panel_estimate` *Linear combination of panel waves, with honest between-wave variance*

Description

Estimates any linear combination `psi = sum(contrast * theta_wave)` of a statistic across panel waves – an annual average (`contrast = rep(1/W, W)`), a net change (`contrast = c(-1, 1)`, i.e. what `change_estimate()` does), a semester contrast, etc. – and its variance `a' Sigma a`, where `Sigma` is the between-wave covariance matrix that the overlap of a rotating panel induces. That covariance is taken straight from the coordinated replicates of `wave_bootstrap()` or `wave_jackknife()`, so it is correct without any analytic covariance formula. Treating the waves as independent (the diagonal of `Sigma` only) understates the variance of an average and overstates that of a change.

Usage

```
panel_estimate(
  wb,
  statistic,
  contrast = NULL,
  waves = NULL,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)
```

```
panel_mean(
  wb,
  variable,
  contrast = NULL,
  waves = NULL,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)

panel_total(
  wb,
  variable,
  contrast = NULL,
  waves = NULL,
  level = 0.95,
  ci_type = c("normal", "t"),
  df = NULL
)
```

Arguments

wb	a weightflow_wave_boot from wave_bootstrap() or a weightflow_wave_jack from wave_jackknife().
statistic	a function function(w, data) returning one number, evaluated per wave.
contrast	numeric weights, one per wave, defining the combination; defaults to the equal-weight average rep(1/W, W). May be named by wave label (matched to waves).
waves	optional character vector of the waves to combine (defaults to all, in order).
level	confidence level for the normal-approximation interval.
ci_type	"normal" (default, z) or "t" (Student t with df df).
df	degrees of freedom for the "t" interval; NULL uses the object's design df.
variable	name of a numeric variable present in every wave.

Value

an object of class weightflow_panel_estimate with estimate, se, V, Vind (the independence-assuming variance), deff = V / Vind, the covariance matrix Sigma, the per-wave point estimates, the contrast, and the confidence interval.

See Also

change_estimate(), wave_bootstrap(), wave_jackknife()

Examples

```
waves <- lapply(1:3, function(t)
  weighting_spec(subset(panel_ine, wave == t & disposition == "R"), base_weights = pw))
names(waves) <- c("T1", "T2", "T3")
wb <- wave_bootstrap(waves, replicates = 100, strata = "stratum", psu = "psu",
                     seed = 1, progress = FALSE)
rate <- function(w, d) weighted.mean(d$unemployed, w, na.rm = TRUE)
panel_estimate(wb, rate)                     # average unemployment level over the waves
panel_estimate(wb, rate, contrast = c(-1, 0, 1))  # T3 - T1 change
```

`panel_merge` *Build the wide longitudinal file from per-wave surveys*

Description

Joins a named list of per-wave `data.frames` on a stable unit key into the wide, one-row-per-unit file that the longitudinal recipe consumes. Each wave's non-key columns are suffixed with the wave name, and per-wave presence indicators `.wf_in_<wave>` (and, if `responded` is given, response indicators `.wf_resp_<wave>`) are added so a later `step_attrition()` can model the response pattern.

Usage

```
panel_merge(
  waves,
  by,
  responded = NULL,
  require = c("any", "all"),
  suffix = "_"
)
```

Arguments

`waves`	a **named** list of per-wave `data.frames`; the names become the wave labels and column suffixes.
`by`	one or more column names (a character vector) forming the unit key, present in every wave – e.g. `"ID"` for a household or `c("ID", "person_no")` for a person.
`responded`	optional string: the name of a response indicator present in every wave, used to build `.wf_resp_<wave>`.
`require`	one of `"any"` (union of units, default) or `"all"` (intersection).
`suffix`	string inserted before the wave label when renaming columns (default `"_"`), e.g. `age` in wave `T1` becomes `age_T1`.

Details

`require = "any"` (the default) keeps every unit seen in at least one wave – which the attrition model needs, because it has to be fitted over responders *and* non-responders. `require = "all"` keeps only the intersection (CEPAL's `s(2) = s1 cap s2 cap ...`); prefer to reach it with `step_attrition()` after estimating, not by dropping units before the model can see them.

Value

a wide `data.frame`, one row per unit, ready for `panel_design()` / `weighting_spec()`.

See Also

`panel_design()`

Examples

```
# reshape two waves of the long panel into one wide, one-row-per-unit file
wide <- panel_merge(
  list(T1 = subset(panel_ine, wave == 1), T2 = subset(panel_ine, wave == 2)),
  by = c("household_id", "person_no"), require = "any")
names(wide)                       # per-wave columns are suffixed _T1 / _T2
```

`panel_pr` *Panel-selection probability for a set of combined waves*

Description

Returns `Pr(panel selection)` for combining `waves` in a `panel_design()` that carries a `rotation_group`: the fraction of rotation-group cohorts present in *all* the combined waves. Its reciprocal is the CEPAL panel base-weight factor used by `step_panel_overlap()` (`prob = panel_pr(pd, c("T1","T2"))`). Surveys without a public rotation-group variable (e.g. the Chilean ENE) cannot use this – derive the probability another way and pass it to `step_panel_overlap()`.

Usage

```
panel_pr(object, waves = NULL)
```

Arguments

`object`	a `wf_panel_design` (from `panel_design()`).
`waves`	optional character vector of the combined waves; defaults to all.

Value

a single probability in `(0, 1]`.

See Also

`panel_design()`, `step_panel_overlap()`

Examples

```
pd <- panel_design(panel_ine, unit = c("household_id", "person_no"), wave = "wave",
                   rotation_group = "rotation_group")
panel_pr(pd)                      # over all waves
panel_pr(pd, c("1", "2"))         # combining waves 1 and 2
```

plot.prepped_weighting_spec *Diagnostic plots for the weights*

Description

Draws the weighting cascade: one histogram of the adjustment factor per step, plus a four-panel summary of the final weights, the cumulative factor, base against final weight, and the design effect by stage. Base graphics only, no dependencies. Use it after `prep()` to see *how* the weights moved, where summary() tells you *by how much*.

Usage

```
## S3 method for class 'prepped_weighting_spec'
plot(x, type = c("all", "factors", "summary"), ...)
```

Arguments

x	a prepped object (output of prep()).
type	"all" (default): per-step adjustment-factor histograms PLUS the summary panel (final weights, cumulative factor, base vs final, deff by stage), all in one grid. "factors": only the per-step factor histograms. "summary": only the summary panel.
...	ignored.

Value

Invisibly, the prepped object `x`. Called for its side effect of drawing the diagnostic plots described above.

Examples

```
fitted <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  prep()
plot(fitted)
```

population *Synthetic target population (sampling frame)*

Description

The simulated frame every weightflow example draws from: 4,495 persons nested in 1,882 households, in 120 primary sampling units, in 4 regions used as strata. Because the whole population is observed, it supplies the known population totals a calibration step needs, and the true values against which a weighted estimate can be checked.

Usage

```
population
```

Format

A data frame with one row per person:

person_id individual identifier

household_id household identifier (cluster)

psu primary sampling unit (segment) within the stratum

region stratum: North, South, East or West

sex F or M

age age in years (18-95)

income annual income

employed employment indicator (0/1)

---

prep *Estimate the weighting cascade*

Description

Runs an inert `weighting_spec()` recipe. Starting from the design base weights, `prep()` applies each step in the order it was piped, multiplying the current weight by that step's adjustment factor, and returns an object holding the weight at every stage, the per-step diagnostics and the quality alerts. This is the only function that computes weights.

Usage

```
prep(spec, min_cell_n = 30, max_factor = 2.5, warn = FALSE)
```

Arguments

`spec`	a weighting_spec.
`min_cell_n`	integer. Minimum number of cases per adjustment cell (weighting class, post-stratum). Cells below this raise a (non-fatal) warning recommending collapsing or switching to raking. Default 30, following Kalton and Flores-Cervantes (2003). Set to NULL to disable.
`max_factor`	numeric. Adjustment factor above which a cell is flagged as excessive. Default 2.5. Set to NULL to disable.
`warn`	logical. If TRUE, the quality alerts are also raised as R warnings during prep(). Default FALSE: alerts are always computed, stored on the object (`$alerts`) and shown in the HTML report, but not raised as warnings, so they do not flood bootstrap/jackknife replicate fits.

Value

a "prepped_weighting_spec" object. Every quality incident is recorded in `$alerts` (readable with `weighting_alerts()` / `has_alerts()`), regardless of `warn`. This includes warnings a step raises internally, such as a calibration that could not meet its constraints: they are captured into `$alerts` even when the surrounding warnings are suppressed, so `$alerts` is the single reliable channel for programmatic quality control.

See Also

weightflow-alerts for the catalogue of quality alerts prep() can raise, `weighting_alerts()` / `has_alerts()` to read them, and `vignette("inspecting-auditing")` for the full quality-control workflow.

Examples

```
rec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region")
prep(rec)
```

---

print.weightflow_boot *Print a bootstrap replicate-weight object*

Description

Compact one-screen summary of a `weightflow_boot` object: how many replicates were requested, how many units are in the data, how many of them are still active (final weight above zero), and which columns defined the resampling design.

Usage

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

Arguments

`x`	a `weightflow_boot` object.
`...`	ignored.

Value

(invisibly) the object.

print.weightflow_jack *Print a jackknife replicate-weight object*

Description

Compact one-screen summary of a `weightflow_jack` object: how many delete-a-PSU replicates were built, how many units are in the data, how many of them are still active (final weight above zero), and which columns defined the deletion design.

Usage

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

Arguments

`x`	a `weightflow_jack` object.
`...`	ignored.

Value

(invisibly) the object.

read_recipe *Read a weighting recipe from a YAML file*

Description

Reads a recipe written by `write_recipe()`. With `data = NULL` (the default) it returns an inspectable recipe manifest (for review or archival); pass `data` to reconstruct an executable `weighting_spec` bound to that data, ready for `prep()`.

Usage

```
read_recipe(file, data = NULL, references = NULL, allow_code = FALSE)
```

Arguments

`file`	path to the recipe `.yml`/`.yaml` file.
`data`	optional data frame. When supplied, the recipe is rebuilt into a `weighting_spec` on this data. The columns the steps reference (weights, auxiliaries, disposition flags) must exist in `data`.

`references` optional named list, named by the id of the step that needs each object, restoring what the recipe stores only as a descriptor because it is microdata rather than metadata: a `reference_sample()` for a step that calibrates or pseudo-weights against a reference, and the `population` data frame of `step_calibrate()` / `step_model_calibration()`.

`allow_code` single logical, the gate on **every** executable thing a recipe file can carry: the conditions and formulas it stores as text, and a step that stores an R function as source. With `FALSE` (the default) a stored expression is accepted only if every call in it is on a fixed whitelist of data-manipulation functions (comparisons, arithmetic, `is.na()`, `factor()`, `cut()`, string and `apply`-free helpers), and a function node is refused outright. That matters because a recipe is meant to be exchanged between organizations: an expression like `{ system("...") ; responded }` runs at the first `prep()`, not at read time. Set `TRUE` only for a file you trust, exactly as you would `source()` it.

Details

A reconstructed recipe is validated only when you call `prep()`: a hand-edited recipe with an out-of-range or mistyped value surfaces its error there, not at `read_recipe()` time. And because reading a recipe evaluates the stored expressions, only read recipes you trust, as you would `source()` an R script.

Value

With `data = NULL`, a `weightflow_recipe` manifest (a list with a print method). With `data`, a `weighting_spec`.

See Also

`write_recipe()`

Other recipe serialization: `write_recipe()`

Examples

```
spec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region")
f <- tempfile(fileext = ".yml"); write_recipe(spec, f)
read_recipe(f)                       # inspect the manifest
spec2 <- read_recipe(f, data = sample_survey)   # rebuild an executable recipe
```

---

reference_sample *Use a weighted survey as the calibration reference instead of a frame*

Description

Wraps a reference-survey microdata `data.frame` together with its design weights so it can be passed as the `population` argument of `step_model_calibration()` (and any step that takes a `population` frame). The calibration totals are then the *weighted* sums over the reference survey – an estimate of the population totals – instead of unweighted sums over a full frame. This is the model-assisted / two-survey setup: fit the model on your sample, project it onto a larger reference survey, and calibrate to the weighted totals of the projection (Wu and Sitter 2001; Kim and Rao 2012).

Usage

```
reference_sample(data, weights, replicates = NULL)
```

Arguments

`data`	a `data.frame` of reference-survey microdata, with the columns used in `x_formula` and the model predictors.
`weights`	either the name (string) of a positive weight column in `data`, or a numeric vector with one weight per row.
`replicates`	optional numeric matrix (or data.frame) of replicate weights for the reference survey – one row per reference unit, one column per replicate – used to propagate the reference sampling variance through `bootstrap_weights()`. `NULL` (default) treats the totals as fixed. Note that only `bootstrap_weights()` pairs the reference replicates and propagates this variance; `jackknife_weights()` treats the estimated totals as fixed even when `replicates` is supplied, so use the bootstrap when this component matters.

Details

A reference survey with all weights equal to 1 reproduces the plain-frame behaviour exactly. To propagate the reference survey's own sampling variance into the recipe-aware bootstrap, pass its replicate weights through `replicates`: each bootstrap replicate then re-estimates the totals from the paired reference replicate (Opsomer and Erciulescu 2021), so the extra variance from estimating the totals is captured. Without `replicates` the totals are treated as fixed – the same assumption made when calibrating to another survey's published totals.

How much that assumption costs depends on what the reference is used for, and the two uses are not equivalent. In `step_model_calibration()` the reference supplies *totals*, a smooth functional of it. In `step_pseudoweight()` every reference unit is a row of the pooled participation fit, so the fitted propensities – and therefore every pseudo-weight – depend on the reference unit by unit. Treating it as fixed there understates the variance even when the reference is several times the size of the sample, and the recipe cannot repair this on its own: building the reference's replicates needs its strata, PSUs and finite-population correction (and its control totals, if it was calibrated), which only the reference's own producer has. So pass them, from the survey's published replicate file or built with `bootstrap_weights()` on the reference's design; without them `bootstrap_weights()` and `jackknife_weights()` say which assumption they fell back on.

Value

data tagged so that step_model_calibration() weights its totals by weights. It is still an ordinary data.frame.

See Also

step_model_calibration()

Examples

```
ref <- reference_sample(population, weights = rep(1, nrow(population)))
```

---

report_panel *Panel / longitudinal HTML report*

Description

Builds an HTML report by ADDING the panel cards to the standard report. When a longitudinal weight is given, it renders the full report_weighting() page for that weight (cascade, weight distribution, deff, ...) and injects a "Panel / longitudinal" section with the rotation structure (panel_design()), the coordinated net-change variance (change_estimate()), the attrition/retention summary, and the gross flows (transition_matrix() / boot_transition()). Without a longitudinal weight it writes a small standalone page with those cards. Any argument may be NULL; its card is skipped.

Usage

```
report_panel(
  design = NULL,
  change = NULL,
  longitudinal = NULL,
  transition = NULL,
  coordinated = NULL,
  estimates = NULL,
  variance = NULL,
  file = NULL,
  open = TRUE,
  lang = c("en", "es")
)
```

Arguments

design	a wf_panel_design from panel_design().
change	a weightflow_change from change_estimate() / change_mean().
longitudinal	a prepped longitudinal weighting_spec; when given, the report is the full weighting report of that weight with the panel section added.

transition	a weightflow_transition / _boot from the flow functions.
coordinated	the coordinated wave_bootstrap() / wave_jackknife() object behind change, so the report shows a "Replication-based variance estimation" card stating the coordinated method, replicates, strata, PSUs, recipe-aware setting and seed.
estimates	results of the estimation grammar: a weightflow_estimation_result from collect_estimates(), an uncollected step_estimate() pipeline (collected here), or a named list of either – the names become the card titles. Each becomes a results block with the levels behind a change, its coordinated standard error and confidence interval, a dot-and-whisker chart across the step_domain() cells, and the subpopulation any step_filter() restricted it to.
variance	the longitudinal-weight bootstrap_weights() / jackknife object, so the base report shows the variance/replication card (deff, method, replicates) for the longitudinal weight instead of "replicate weights not created". The coordinated CHANGE variance is shown by the change card.
file	output path; a temporary .html if NULL.
open	open the file in a browser.
lang	"en" (default) or "es".

Value

the path to the written HTML file, invisibly.

See Also

report_weighting(), panel_design(), change_estimate(), boot_transition()

---

report_weighting *Self-contained HTML quality report for a weighting recipe*

Description

Writes a single, self-contained HTML file documenting how a prepped recipe turned the design base weights into the final weights: the cascade, the parameters requested at each step, the per-stage weight summary, the step-specific diagnostics (calibration, nonresponse, trimming), the fieldwork outcome rates and an audit trail. It is the deliverable to attach to a methodological report or a weighting annex: everything is inline, so the file opens offline and can be archived or emailed as one artifact.

Usage

```
report_weighting(
  object,
  file = NULL,
  open = TRUE,
  plots = TRUE,
```

```
  narrative = TRUE,
  lang = c("en", "es"),
  metadata = NULL,
  replicates = NULL,
  domains = NULL,
  y_vars = NULL
)
```

Arguments

object	a prepped object (output of prep()).
file	output path; if NULL, a temporary .html file.
open	logical; open the file in the browser.
plots	logical; add per-step plots (weight before-vs-after scatter and adjustment-factor histogram), drawn as self-contained inline SVG (no graphics device or extra package required).
narrative	logical; add an auto-generated methodological narrative – an executive summary at the top and a natural-language paragraph on each step explaining what was done and why (built from the step's own parameters and diagnostics), in the spirit of a GSBPM / ESQRS methodological report.
lang	language of the narrative: "en" (default) or "es".
metadata	optional named list of reference metadata (SIMS / ESMS concepts) shown as a header card, e.g. survey, reference_period, geography, producer, author, contact, frame, totals_source, totals_date, version, confidentiality, notes. Recognised keys get a proper label; any other key is shown as given. survey is also woven into the executive summary. totals_source/totals_date document where the calibration control totals come from and their reference date.
replicates	optional weightflow_boot or weightflow_jack object (from bootstrap_weights() / jackknife_weights()). If given, a "Replication design for variance" card documents the method, number of replicates, strata / PSU structure, lonely-PSU handling, seed, cores and run time, and warns when few PSUs per stratum favour JKn.
domains	optional one-sided formula of grouping variables for a per-domain reliability card. Each term becomes one table (+ = separate tables, : = crossed), showing the active n, sum of weights, CV, Kish design effect and effective sample size within each domain. E.g. domains = ~ region + region:sex.
y_vars	optional character vector of survey outcome variables. When a nonresponse-by-calibration step is present, the auxiliary-quality table adds, for each auxiliary, its weighted correlation with each y among respondents (Sarndal-Lundstrom criterion (ii): a good auxiliary also explains the y).

Value

(invisibly) the path to the HTML file.

Examples

```
fitted <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  prep()
# writes a self-contained HTML report to a temporary file (open = FALSE so
# nothing is launched); use open = TRUE to view it in the browser.
path <- report_weighting(fitted, open = FALSE)
```

---

sample_one *Synthetic address sample with one selected person per household*

Description

A multistage sample of 417 addresses (stratum, then PSU, then household, then one person inside the reached household) carrying every complication a household survey meets: unresolved eligibility, out-of-scope addresses, household nonresponse, unequal within-household selection and person nonresponse. It is the dataset that exercises the complete weighting cascade.

Usage

```
sample_one
```

Format

A data frame with one row per sampled household (the selected person, or a single placeholder row for non-roster cases):

person_id, household_id, psu identifiers

region stratum

sex, age selected person's attributes (NA on non-roster rows)

pw design base weight (product of the stage selection probabilities)

status "eligible", "ineligible" or "unknown"

disposition full field disposition as a single factor (a recode of the indicator columns): "eligible respondent", "eligible nonrespondent", "household nonresponse", "ineligible" or "unknown eligibility"

unknown_elig 1 if eligibility is unknown (no roster)

ineligible 1 if the address is out of scope (no roster)

hh_responded 1 reached, 0 household nonresponse, NA for non-eligible

responded 1 if the selected person responded (NA on non-roster rows)

n_elig number of eligible persons in the household (NA on non-roster rows)

p_within within-household selection probability of the selected person

income, employed survey outcomes; NA unless the person responded

sample_survey *Synthetic person sample with a take-all household roster*

Description

A stratified two-stage sample of 467 persons drawn from population: PSUs within region, then households within PSU, then **every** adult of the selected household (take-all roster). It carries unequal design base weights, an unknown-eligibility flag and a person-level response indicator, and it is the dataset the short examples in this package use.

Usage

```
sample_survey
```

Format

A data frame with one row per sampled person:

person_id, household_id, psu identifiers

region, sex, age frame auxiliaries, known for all units

pw design base weight (inverse sampling fraction)

unknown_elig 1 if eligibility is unknown

responded 1 if the person responded

income, employed survey outcomes; NA for nonrespondents

step_assert *Assert quality conditions on the weights*

Description

A checkpoint that leaves the weights untouched and instead verifies that they meet quality thresholds at this point of the cascade, raising an error or a warning when they do not. Use it to stop a production pipeline before bad weights are published, in the spirit of a validation step inside a recipe.

Usage

```
step_assert(
  spec,
  max_deff = NULL,
  max_weight_ratio = NULL,
  min_n_eff = NULL,
  on_fail = c("error", "warning"),
  id = NULL
)
```

Arguments

spec	a weighting_spec.
max_deff	numeric or NULL. Maximum acceptable Kish design effect.
max_weight_ratio	numeric or NULL. Maximum allowed final/base weight ratio (per active unit).
min_n_eff	numeric or NULL. Minimum acceptable effective sample size.
on_fail	"error" (stop the cascade) or "warning".
id	optional string: a stable identifier for this step, shown in the recipe print-out; defaults to a derived "<class>_<k>".

Value

The input weighting_spec with this checkpoint appended to its recipe. The check is recorded only; it is evaluated when prep() is called and does not modify the weights.

See Also

Other weighting steps: step_calibrate(), step_cre(), step_drop_ineligible(), step_model_calibration(), step_nonresponse(), step_nr_sensitivity(), step_pseudoweight(), step_rescale(), step_round(), step_select_within(), step_subsample(), step_trim(), step_trim_calibrated(), step_trim_weights(), step_unknown_eligibility()

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_assert(max_deff = 5, on_fail = "warning") |> prep()
```

---

step_attrition *Attrition adjustment for panel waves*

Description

The panel-facing nonresponse step: it adjusts for **attrition** (nonresponse between waves) when building a longitudinal weight, reweighting the units that stayed in to also represent those that dropped out, among the units that remain **eligible**. It is a thin wrapper over step_nonresponse() – so the theory stays visible in the recipe – that uses the same estimators but under a name that reads correctly in a panel cascade and with the panel conventions: the covariates should come from a wave where the unit was observed (e.g. the first period), and it does NOT absorb eligibility (out-of-scope and unknown-eligibility go in step_drop_ineligible() / step_unknown_eligibility()).

Usage

```
step_attrition(
  spec,
  respondent,
  method = c("propensity", "rhg", "weighting_class", "calibration"),
  formula = NULL,
  by = NULL,
  engine = c("logit", "tree", "forest", "boost"),
  num_classes = 5L,
  weight_model = TRUE,
  cluster = NULL,
  crossfit = NULL,
  crossfit_seed = NULL,
  totals = NULL,
  count = NULL,
  calfun = c("linear", "logit", "raking"),
  bounds = NULL,
  penalty = NULL,
  equal_within_cluster = FALSE,
  maxit = 50L,
  tol = 1e-06,
  id = NULL
)
```

Arguments

spec	a weighting_spec.
respondent	an unquoted 0/1 column or logical condition, TRUE for the units that responded in the wave being adjusted (e.g. `disposition_T2 == "R"`).
method	attrition estimator: `"propensity"` (individual `1/phi`, the SLID/ECLAC response-propensity weighting; default), `"rhg"` (response homogeneity groups: propensity stratified into `num_classes` classes), `"weighting_class"` (design-variable cells), or `"calibration"` (Sarndal-Lundstrom).
formula	model formula for `"propensity"`/`"rhg"`, in covariates observed for responders and nonrespondents (from a wave where the unit was seen).
by	adjustment cells for `"weighting_class"`.
engine	propensity engine (`"logit"`/`"tree"`/`"forest"`/`"boost"`).
num_classes	number of propensity classes for `"rhg"`.
weight_model	logical. Only for method = "propensity": whether to fit the response-propensity model with the incoming weights (`TRUE`, the default) or unweighted (`FALSE`). Fitting unweighted can reduce the variance of the propensity estimates when the weights are unrelated to response given the model covariates, at the cost of possible bias if they are (Little & Vartivarian 2003). The 1/p (or class) adjustment always uses the design weights; only the model fit is affected.
cluster	character or NULL. If given, the adjustment is done at the cluster level for whole-cluster nonresponse: each cluster counts once with its (uniform) weight;

in "weighting_class" the redistribution is between responding and nonresponding clusters within the cells, and in "propensity" the model is fitted with one row per cluster (cluster auxiliaries), predicting the cluster's response. The resulting factor is assigned to every member; nonresponding clusters go to zero. As always, only active units (weight > 0) take part, so units already dropped (unknown eligibility, ineligible) are excluded automatically. For `method = "calibration"`, `cluster` is used together with `equal_within_cluster = TRUE` for integrative (one weight per cluster) calibration.

The cluster need not be a household: it is any grouping whose members share a fate and a weight – a dwelling, an area segment, or a whole primary sampling unit (an entire PSU inaccessible, then redistributed within its stratum). A methodological consequence to keep in mind: a cluster-level adjustment preserves the *mass of clusters* in each cell (the cluster weight is the mean of its members, in the sense of Valliant et al. 2018), and the factor is uniform within the cell; it does **not**, by construction, preserve the mass of the underlying units (persons). That is the job of the later calibration, whose margins bring the person totals back exactly.

`crossfit` integer or NULL. If given (number of folds K >= 2), the propensity is estimated by K-fold cross-fitting: for each fold the model is trained on the other folds and used to predict the held-out fold, so each unit's propensity comes from a model that did not see it. This avoids the overfitting that flexible engines (forest, boost) can produce, which would otherwise inflate the weights. Folds are formed by `cluster` when given (so correlated units stay together). NULL (default) fits and predicts in-sample. For flexible learners it also keeps the design-based variance honest: same-sample predictions can understate the variance even under recipe-aware replication (Dagdoug, Goga and Haziza 2023), so cross-fitting is recommended whenever `engine` is not "logit".

`crossfit_seed` integer or NULL. Seed for reproducible fold assignment when `crossfit` is used.

`totals` (method = "calibration") calibration targets. NULL (default) calibrates the respondents to the R+NR design-weighted totals of `formula` at that stage (the two-phase / sample-level case; Sarndal & Lundstrom 2005); a named vector or a tidy `totals`/`count` input (as in `step_calibrate()`) calibrates to population totals instead.

`count` (method = "calibration", tidy `totals`) string naming the counts column of the totals data frame(s).

`calfun` (method = "calibration") distance function for the calibration factor: "linear", "raking" or "logit", as in `step_calibrate(method = "linear")`.

`bounds` (method = "calibration") numeric c(L, U) with L < 1 < U. Bounds on the calibration factor, to keep the nonresponse factors positive.

`penalty` (method = "calibration", unbounded) NULL or positive cost(s) for ridge (penalized) calibration.

`equal_within_cluster`
(method = "calibration") logical. If TRUE, integrative (Lemaitre-Dufour) nonresponse calibration: the responding members of a household (`cluster`) share a single calibration factor, so the adjustment keeps the weights constant within household. Requires `cluster`. FALSE (default) calibrates each responding unit on its own.

`maxit`, `tol` (method = "calibration") convergence control for the bounded or exponential-distance calibration solver.

`id` optional stable step id.

Details

The attrition adjustment prescribed by ECLAC's household-survey manual (ch. XVI) and used by Statistics Canada's SLID (Naud 2002; LaRoche 2003) is **response-propensity weighting** (`method = "propensity"`, i.e. inverse of the estimated response probability; Little 1986; Rosenbaum 1987) – not an ECLAC invention. `method = "rhg"` is the response-homogeneity-group variant (propensity stratified into `num_classes` groups, then the class-mean rate), which stabilises the weights.

Two ECLAC prescriptions are **not** implemented, and the difference matters when they apply. The manual (ch. XVI, sec. B.1.b) allows two fallback imputations for units with no auxiliary information at all: a nonrespondent whose rotation panel does not overlap (impute the overall effective response rate as its `phi`), and a newly incorporated nonrespondent (impute the adjusted expansion factor of its household). Here a covariate that is `NA` for an eligible unit is an **error**, not an imputation – an `NA` propensity would let that nonrespondent survive the adjustment silently, which is the failure the error exists to prevent, and the package will not silently substitute a value of its own for a missing input.

So when the error fires, the fix belongs in the data, and either ECLAC route is available to you there: impute the missing covariates before the step (the manual's own fallback is the overall effective response rate, i.e. a constant, which is what a model with no covariates for those units amounts to), give a newly incorporated nonrespondent its household's adjusted factor, or restrict `formula` to a covariate set observed for every eligible unit. What the package will not do is choose one of those for you and leave no trace in the recipe. Multi-wave retention chaining (decomposing Pr(in at T3) into Pr(reach T2) x Pr(T2 to T3)) is likewise absent: this step adjusts one transition at a time, which is what ch. XVI specifies for two consecutive periods.

Value

the `weighting_spec` with the attrition step appended.

The arguments of step_nonresponse()

Attrition *is* nonresponse over waves, so this step delegates to `step_nonresponse()` and takes its arguments too. That was not true before: the twelve remaining ones were missing, including `crossfit` – which the step's own quality alert recommends when a flexible engine overfits the propensity, a remedy that could not be applied – and `totals` / `calfun` / `bounds`, i.e. the whole Sarndal-Lundstrom calibration route with nothing to configure.

See Also

`step_nonresponse()`, `step_panel_overlap()`, `step_drop_ineligible()`

Examples

```
wide <- panel_merge(
  list(T1 = subset(panel_ine, wave == 1), T2 = subset(panel_ine, wave == 2)),
  by = c("household_id", "person_no"), require = "all")
```

```
weighting_spec(wide, base_weights = pw_T1) |>
  step_panel_overlap(prob = 5 / 6) |>
  step_drop_ineligible(disposition_T2 == "OS", reason = "left the target population") |>
  step_attrition(respondent = disposition_T2 == "R", method = "propensity",
                 formula = ~ age_T1 + sex_T1 + region_T1) |>
  prep()
```

`step_calibrate` *Calibration to population totals*

Description

Adjusts the weights so that the weighted sample reproduces known population totals of auxiliary variables, while moving the incoming weights as little as possible. This is the calibration estimator of Deville and Sarndal (1992), with raking, post-stratification and linear (GREG) calibration, optional bounds on the adjustment factor, ridge relaxation of the targets, domain partitions and one-weight-per-cluster (integrative) calibration.

Usage

```
step_calibrate(
  spec,
  margins = NULL,
  method = c("raking", "poststratify", "linear"),
  formula = NULL,
  totals = NULL,
  count = NULL,
  by = NULL,
  cluster = NULL,
  equal_within_cluster = FALSE,
  calfun = c("linear", "logit", "raking"),
  bounds = NULL,
  maxit = 50L,
  tol = 1e-06,
  penalty = NULL,
  population = NULL,
  id = NULL
)
```

Arguments

`spec`	a weighting_spec.
`margins`	named list (classic format for "raking"/"poststratify"). Each element is a named numeric vector with the target totals per category. E.g.: list(sex = c(M = 5000, F = 5200), region = c(N = 3000, S = 7200)). Still fully supported; for a tidy alternative see `totals` and `count`.

`method` "raking" (IPF, categorical margins), "poststratify" (post-strata: one or more categorical variables crossed) or "linear" (GREG / regression estimator; handles continuous and categorical auxiliaries together).

`formula` (only method = "linear") auxiliary formula, e.g. ~ sex + income. Uses model.matrix; includes the intercept unless you write ~ 0 + ...

`totals` population totals, in one of two forms. Classic (all methods): for "linear" a named numeric vector aligned with the model.matrix columns (including "(Intercept)" = N); for "raking"/"poststratify" use `margins`. Tidy (recommended): a data frame or a named list of data frames/numbers giving the totals in a friendly way, paired with `count`. For "poststratify", a single data frame with one or more category columns plus a counts column. For "raking", a list of data frames, one per margin. For "linear", a named list whose names match the formula terms: a data frame with all categories for each factor, and a single number for each continuous auxiliary; weightflow builds the model.matrix totals internally (you never handle the intercept or dropped reference category).

`count` (tidy `totals` only) string naming the counts column in the totals data frame(s). All other columns are treated as category variables.

`by` (tidy `totals` only) NULL, or a string naming a domain (partition) column. When given, the weights are calibrated **independently within each domain**, each to its own totals (partitioned / domain calibration). The totals tables carry the domain as a column, and each `count` table is split by it; a continuous total becomes a data frame `domain, value` (one total per domain). The domain variable must NOT appear in `formula` / the margins: it is the partition. Composes with `calfun`, `bounds`, `penalty` and `equal_within_cluster`, applied within each domain. NULL (default) calibrates globally, as before.

`cluster` (only method = "linear") name of the cluster id column (e.g. "household"), for equal weights within the cluster.

`equal_within_cluster` (only method = "linear") logical. If TRUE, Lemaitre-Dufour (1987) integrative calibration: a single weight per cluster. Requires `cluster`. Final weights are equal within the cluster provided the incoming weight is also uniform within the cluster. Works with any `calfun` distance ("linear", "raking" or "logit"), with `bounds` and with `by`.

`calfun` (only method = "linear") distance function for the calibration factor g: "linear" (g = 1 + u, closed form), "raking" (g = exp(u), the exponential/multiplicative distance, which keeps the weights positive and still satisfies the constraints exactly) or "logit" (bounded by construction; requires `bounds`). "raking" and "logit" use the iterative Deville-Sarndal solver and work with the integrative option (`equal_within_cluster`) too.

`bounds` (only method = "linear") numeric c(L, U) with L < 1 < U. Bounds on the calibration factor g (g-weights). With "linear" it truncates; with "logit" it is enforced smoothly. Avoids extreme/negative weights without a separate trimming step.

`maxit, tol` convergence control for raking and bounded calibration.

`penalty` (only method = "linear", unbounded) NULL or positive cost(s) for ridge (penalized) calibration. A positive scalar applies the same cost to every constraint; a named vector sets a cost per constraint (matched to the model.matrix columns).

The cost is scale-free: a large value keeps the constraint (near) exact, a small value relaxes it to control extreme weights when there are many auxiliaries. Under ridge the achieved totals no longer match the targets exactly; the diagnostics report the deviation.

`population` (all methods) a `reference_sample()` (or a plain frame) from which the calibration targets are computed, instead of passing `margins` / `totals`. Give `formula` naming the calibration variables; the targets are the design-weighted sums over the reference (raking margins, poststratify cells, or linear model-matrix totals). If the reference carries replicate weights, `bootstrap_weights()` propagates its sampling variance. Not combined with `margins` / `totals` or `by`.

`id` optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived "<class>_<k>".

Details

The calibrated weight is $w_i = d_i g_i$, close to the incoming weights d_i and reproducing the totals $\sum_{i \in s} w_i \mathbf{x}_i = \mathbf{X}$, with the factor g_i minimizing a distance to d_i: linear $g_i = 1 + \mathbf{x}_i'\boldsymbol{\lambda}$ or exponential ("raking") $g_i = \exp(\mathbf{x}_i'\boldsymbol{\lambda})$. The `penalty` (ridge) relaxes the exact constraint to steady extreme weights when the auxiliaries are many or collinear, minimizing

$$\sum_i \frac{(w_i - d_i)^2}{d_i q_i} + \frac{1}{s} \sum_j c_j (\hat{X}_j - X_j)^2,$$

where c_j is the cost of missing constraint j: as $c_j \to \infty$ the constraint is met exactly, as $c_j \to 0$ the weights return to d_i.

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: `step_assert()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
# Raking to population margins
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_calibrate(method = "raking", id = "calib_main",
                 margins = list(sex    = c(table(population$sex)),
                                region = c(table(population$region)))) |>
  prep()
# the id ("calib_main") labels the step in the print-out and selects it in
# collect_step_detail(fit, "calib_main")
```

```
# ridge (penalized) calibration: relaxes the targets to control extreme
# weights; a smaller penalty relaxes more. Uses only base R.
pop_tot <- c("(Intercept)" = nrow(population),
             regionSouth = sum(population$region == "South"),
             regionEast  = sum(population$region == "East"),
             regionWest  = sum(population$region == "West"),
             sexM        = sum(population$sex == "M"))
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_calibrate(method = "linear", formula = ~ region + sex,
                 totals = pop_tot, penalty = 1) |>
  prep()

# --- Tidy `totals` format (recommended) ---------------------------------
# Post-stratification: give the population counts as a data frame with one or
# more category columns plus a counts column named by `count`.
ps_totals <- as.data.frame(table(region = population$region, sex = population$sex))
weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "poststratify", totals = ps_totals, count = "Freq") |>
  prep()

# Raking: a list of data frames, one per margin.
m_region <- as.data.frame(table(region = population$region))
m_sex    <- as.data.frame(table(sex = population$sex))
weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 totals = list(m_region, m_sex), count = "Freq") |>
  prep()

# Linear/GREG with mixed auxiliaries: data frames for categoricals (all
# categories) and a single number for a continuous total. weightflow builds
# the model.matrix totals internally, so you never drop a reference category.
resp <- subset(sample_survey, responded == 1)
weighting_spec(resp, base_weights = pw) |>
  step_calibrate(method = "linear", formula = ~ region + sex + income,
                 totals = list(region = m_region, sex = m_sex,
                               income = sum(population$income)),
                 count = "Freq") |>
  prep()

# Domain (partitioned) calibration: `by` calibrates independently within each
# domain, each to its own totals. The domain is an extra column in the tidy
# totals, not a term in the formula/margins. Here we rake two margins (sex and
# age group) within each region, which a single global cross could not express.
pop  <- transform(population,
  age_grp = cut(age, c(0, 30, 45, 60, Inf), labels = c("18-30","31-45","46-60","60+")))
samp <- transform(sample_survey,
  age_grp = cut(age, c(0, 30, 45, 60, Inf), labels = c("18-30","31-45","46-60","60+")))
sex_by_region <- as.data.frame(table(region = pop$region, sex     = pop$sex))
age_by_region <- as.data.frame(table(region = pop$region, age_grp = pop$age_grp))
weighting_spec(samp, base_weights = pw) |>
  step_calibrate(method = "raking",
                 totals = list(sex_by_region, age_by_region),
```

```
                count = "Freq", by = "region") |>
  prep()
```

`step_cre` *Composite regression estimator (CRE / regression composite estimation)*

Description

Final-weight step for rotating panels that exploits the sample overlap **in the estimation**, not only in the variance. It is the composite calibration of the Canadian LFS (Gambino, Kennedy and Singh 2001; Fuller and Rao 2001) and the composite calibration of the Uruguayan ECH (INE, methodology sec. 8.4): the same estimator. On top of the ordinary calibration to known population totals `X`, it adds a second set of constraints to control totals `Zhat` estimated from the **previous wave** (the composite auxiliaries: previous-month labour status, optionally crossed by domains). Because `Zhat` uses the previous wave's composite weights, the estimator is recursive.

Usage

```
step_cre(
  spec,
  previous = NULL,
  status,
  composite = list(NULL),
  id_unit,
  formula,
  totals = NULL,
  count = NULL,
  birth = NULL,
  alpha = 2/3,
  overlap = "auto",
  rescale_previous = FALSE,
  on_missing_prev = c("carry_backward", "zero"),
  cluster = NULL,
  equal_within_cluster = FALSE,
  calfun = c("linear", "logit", "raking"),
  bounds = NULL,
  rotation_group = NULL,
  n_groups = NULL,
  status_ref = NULL,
  id = NULL
)
```

Arguments

`spec`	a weighting_spec (its recipe up to here yields the nonresponse-adjusted weights `w_nr` that this step starts from).

previous — the prepped previous-wave recipe (prep() output), which supplies the previous composite weights and the previous labour status for Zhat and the overlap imputation. NULL (default) is the **seed** wave: with no t-1 there is no composite block and the step reduces to an ordinary linear calibration to X – the start of the recursion.

status — bare or quoted name of the categorical labour-status column (e.g. employed / unemployed / inactive) in the current wave; must exist with the same name in previous. Its indicators are the composite auxiliaries.

composite — a **list of domain crossings**, each defining one block of composite control totals. NULL = country total; a string = status crossed by that domain (e.g. "sex"); a character vector = status crossed by the interaction (e.g. c("sex","age")). This mirrors the methodologies' composite-variable lists: the ECH's are list(NULL, "sex", "department"). Default list(NULL).

id_unit — character vector naming the unit key that links waves (e.g. c("household","person")). Used to look up each current unit's t-1 status.

formula — the demographic auxiliary formula for x (as in step_calibrate() method = "linear"), e.g. ~ age_sex + region.

totals, count — the population totals X for x, in the same forms accepted by step_calibrate() (a named vector aligned to the model.matrix, or the tidy named list with count).

birth — expression (evaluated in the current data) that is TRUE for the birth rotation group (the units with no t-1 value to carry). If NULL, birth units are those not found in previous by id_unit – which means a key that fails to link is indistinguishable from a genuine entrant, and a broken id_unit then looks like an enormous rotation while the change correction quietly stops working. The step warns when the implied overlap falls below half (or below overlap minus ten points, when overlap is given as a number) and errors below 2%, but the only way to separate the two properly is to declare birth yourself, or to pass the design's nominal overlap so the check has something real to compare against.

alpha — MR1/MR2 mixing constant between 0 and 1. alpha = 0 targets the level only (MR1), alpha = 1 the change only (MR2). Default 2/3 (Chen and Liu 2002).

overlap — the overlap rate used by the MR2 carry-backward correction: "auto" (default) estimates it as the w_nr-weighted overlap fraction, or a number (e.g. 5/6, the nominal LFS/ECH rate).

rescale_previous — put the composite block on the current wave's population scale: Zhat <- Zhat * N_t / Nhat_{t-1} (equivalently, calibrate the composite auxiliaries on proportions). Zhat is a total estimated with the previous wave's weights, so it lives on Nhat_{t-1} while totals fixes this wave's N_t; the identity that keeps the composite block a smoother rather than a level shift assumes the two agree. When they differ, the whole gap is discharged onto the status estimate with every constraint met and converged = TRUE. FALSE (default) keeps the totals as given and warns when the two scales differ by more than 1%; the better fix is to calibrate both waves to the same series of population projections.

on_missing_prev — how to treat non-birth units without a valid t-1 status (new household members, newly of working age): "carry_backward" (default; set z_{t-1} = z_t) or "zero" (z_{t-1} = 0).

`cluster`, `equal_within_cluster`
: integrated method of weighting: with `equal_within_cluster = TRUE` (needs `cluster`) the auxiliaries `x` and `z` are averaged to the cluster so members share one weight (Lemaitre-Dufour 1987).

`calfun`, `bounds`
: distance and g-bounds for the calibration, as in `step_calibrate()`. Use `bounds = c(L, U)` with a logit `calfun` to avoid final weights below 1.

`rotation_group`
: optional name of the rotation-group column. When given, the step adds the **equal-representation** constraints both surveys impose: each rotation group must weight to the same working-age total, `N / G` (ECH sec. 8.4.1, `sum_{i in s_g} w_i = N_PET / 6`; LFS sec. 6.3.1). `N` is read from the intercept target in `totals` and `G` is the number of groups; the last group is left implied (its total follows from the others and `N`), so `G - 1` constraints are added to the demographic block. `NULL` (default) omits them.

`n_groups`
: the number of rotation groups the **design** has, e.g. `6`. Only used with `rotation_group`, and worth giving: `G` is a property of the design, not of who answered this wave, and the target `N / G` is wrong the moment the two differ. If a group has no active units – fully attrited, or a domain with no respondents in it – and `G` is read off the sample, every remaining group is calibrated to `N / (G - 1)`, i.e. a whole group's population shared out among the others, while the intercept still targets `N`. The step refuses that rather than solving it, because the two constraints cannot both hold. When `n_groups` is not given, `G` comes from the column's factor levels (which survive a level going empty); a character column carries no such record, so the step warns that it had to read `G` off the wave.

`status_ref`
: the status level held as reference (dropped from each cell block to avoid the mechanical collinearity between the full status indicators and the demographic block, which pins the same cell totals). The dropped total is implied by the others plus `X`, so the estimator is unchanged; naming the level only sets which one is implicit (e.g. `"inact"` to keep employed/unemployed explicit). `NULL` (default) drops the last level in sorted order. One level is always dropped: keeping every level would make the block collinear with the intercept in `X`.

`id`
: optional stable identifier for this step.

Details

The composite auxiliary `z` is known for the overlap sample (units present in `t-1`) and imputed for the birth rotation group (units entering at `t`, with no `t-1` value) by combining two imputations: MR1 (mean imputation, aimed at the level) and MR2 (carry-backward, aimed at the change), mixed as `z = (1 - alpha) z1 + alpha z2` with `alpha = 2/3`.

The composite weights are `w_cre = w_nr * g`, with `g` the GREG factor of the augmented calibration `[x | z]` to `[X | Zhat]`; the same solver as `step_calibrate()`. With `equal_within_cluster = TRUE` the auxiliaries are averaged to the household (integrated method of weighting, Lemaitre-Dufour 1987) so every household member shares one final weight.

Value

the input `weighting_spec` with the CRE step appended (evaluated at `prep()`). Chain waves by passing each prepped wave as the `previous` of the next.

References

Gambino J, Kennedy B, Singh MP (2001). Regression composite estimation for the Canadian Labour Force Survey. *Survey Methodology* 27(1):65-74. Fuller WA, Rao JNK (2001). A regression composite estimator with application to the Canadian Labour Force Survey. *Survey Methodology* 27(1):45-51. Instituto Nacional de Estadistica (Uruguay). Metodologia de la Encuesta Continua de Hogares, seccion 8.4 (calibracion compuesta).

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
# Composite (CRE) estimation on the 6-month rotating panel `panel_ine`.
# `lf_status` is the previous-wave labour status (emp / unemp / inact).
t1 <- subset(panel_ine, wave == 1 & disposition == "R")
t2 <- subset(panel_ine, wave == 2 & disposition == "R")
t1$sex <- factor(t1$sex); t2$sex <- factor(t2$sex)

# ONE population vector for both waves, as a series of projections would be.
# `Zhat` is a total on the previous wave's scale, so calibrating each wave to its
# own design-weighted total puts two population scales in one system and the
# difference lands on the status estimate (see `rescale_previous`).
Xpop <- colSums(t1$pw * stats::model.matrix(~ sex, data = t1))

# seed wave: no previous month, so step_cre() reduces to a linear calibration to X
seed <- weighting_spec(t1, base_weights = pw) |>
  step_cre(previous = NULL, status = lf_status, formula = ~ sex,
           totals = Xpop, status_ref = "inact") |>
  prep()

# composite wave: augment X with the previous-wave status, country-level and by sex
fit2 <- weighting_spec(t2, base_weights = pw) |>
  step_cre(previous = seed, status = lf_status, composite = list(NULL, "sex"),
           id_unit = c("household_id", "person_no"), formula = ~ sex, totals = Xpop,
           alpha = 2/3, status_ref = "inact") |>
  prep()
fit2
```

`step_cross_sectional` *Declare the recipe's scope: cross-sectional or longitudinal weights*

Description

Two argument-free declarative steps that say what the recipe is building. They do not touch the weights (like `step_assert()`); they record the intent so the rest of the recipe, the report and the estimators behave accordingly, and so a wrong-purpose estimate can be flagged. All the panel detail – waves, rotation group, reference wave – already lives in `panel_design()`, so these steps need no arguments.

Usage

```
step_cross_sectional(spec, id = NULL)

step_longitudinal(spec, id = NULL)
```

Arguments

`spec`	a weighting_spec.
`id`	optional string identifier for the step.

Details

Put the scope step first. `step_cross_sectional()` is the default (a plain recipe with no scope step behaves as cross-sectional). `step_longitudinal()` requires the data to be tagged by `panel_design()` and marks the recipe as building the panel (longitudinal) weight, so steps like `step_panel_overlap()` apply and the calibration targets the reference wave.

Value

The input `weighting_spec` with the scope declared.

ECLAC conventions for the longitudinal weight

The three points below are conventions of ECLAC's household-survey manual (ch. XVI). The package cannot enforce them – a vector of control totals carries no label saying which period it belongs to – so they are the reader's to apply.

Who is in the longitudinal population. It is made up of the units that were in the target population at the *first* period **and remained in it** through the last. Units that left (death, migration, institutionalization) are out, and so are units that *entered* after the first period: a panel adds no elements over time. Drop the first group with `step_drop_ineligible()` before the attrition step – leaving the target population is not nonresponse and must not be compensated by redistributing weight.

Calibrate to the FIRST period's totals. ECLAC is explicit that the auxiliary totals used in the calibration must represent the population of the *first* period of interest, because a panel that adds no elements over the measurement periods is representative only of the period in which it was selected (ch. XVI, sec. B.2.c). In the original:

```
"los totales auxiliares utilizados en la calibracion deben representar la
 poblacion del primer periodo de interes, puesto que, al conformar un panel
 que no suma elementos a lo largo de los periodos de medicion, la muestra
 sera representativa unicamente del periodo en el que fue seleccionada"
```

Passing the *later* period's projections is a silent error: the recipe converges, the totals close, and the weights represent a population the sample never had a chance to cover.

Cross-sectional estimates from a longitudinal file are reference only. They will not match the published cross-sectional figures **and should not**: the target population of the panel combination is not the target population of the cross-section (ch. XVI, sec. C). Use the longitudinal weight for gross flows, transitions and durations – what it exists for – and the cross-sectional weight for levels.

See Also

`panel_design()`, `step_panel_overlap()`

Examples

```
wide <- panel_merge(
  list(T1 = subset(panel_ine, wave == 1), T2 = subset(panel_ine, wave == 2)),
  by = c("household_id", "person_no"), require = "all")
weighting_spec(wide, base_weights = pw_T1) |>
  step_longitudinal() |>
  step_panel_overlap(prob = 5 / 6) |> prep()
```

`step_domain` *Declarative estimation over a coordinated panel object*

Description

`step_domain()` and `step_estimate()` build an estimation pipeline (`weightflow_estimation`) on top of a saved `wave_bootstrap()` / `wave_jackknife()` object, so that changes, levels and linear contrasts – with the honest overlap covariance – can be requested declaratively and disaggregated, the way the weighting recipe is built with `step_*`. It is a thin front end over `change_estimate()`, `panel_estimate()` and `level_estimate()`; the coordinated replicates do the variance. Evaluate with `collect_estimates()` (printing does it too).

Usage

```
step_domain(x, ...)

step_filter(x, condition)

step_estimate(
  x,
  statistic,
  over = NULL,
  type = c("absolute", "relative"),
  waves = NULL,
  contrast = NULL,
  level = 0.95,
  label = NULL
```

```
)

step_transition(x, from, to, format = c("row", "col", "joint", "counts"))
```

Arguments

x	a weightflow_wave_boot / weightflow_wave_jack, or a weightflow_estimation to extend.
...	for step_domain(), one or more grouping columns (unquoted or as strings); they stack, so the disaggregation is their cross (e.g. region x sex). A column may not be named after one of the result table's own columns (estimand, over, type, estimate, se, ci_lower, ci_upper, rho); rename it in the wave data first.
condition	for step_filter(), a logical expression (unquoted) that selects the subpopulation to estimate over – e.g. age >= 25 & age <= 54. It is evaluated in each wave's data; rows are **masked** (not dropped), so the coordinated replicate structure and the overlap covariance are preserved. Multiple step_filter() calls stack (their conditions are ANDed).
statistic	the estimand, as a DSL call – mean(var), total(var), prop(cond), ratio(num, den), quantile(var, p) – or a function(w, data). Every argument is evaluated in the wave data and must give one numeric or logical value per unit: prop() takes a **condition** (prop(status == "unemployed")), not a category, and p in quantile() is a probability in [0, 1] (0.5, not 50). ratio() is taken over the domain where numerator **and** denominator are both observed, as survey::svyratio(na.rm = TRUE) is. The weighted quantile is R's type 5.
over	what to estimate: "change" (between two waves), "level" (one wave) or "contrast" (a linear combination, with contrast=). Default: "change" if the object has >= 2 waves, else "level"; "contrast" is implied when contrast= is given.
type	for over = "change", "absolute" (default) or "relative" (theta2/theta1 - 1).
waves	optional wave label(s): one for "level", two for "change".
contrast	numeric weights (one per wave) for over = "contrast".
level	confidence level.
label	optional name for the estimand (defaults to the statistic's expression).
from, to	for step_transition(), the from-/to-state columns.
format	transition table format ("row", "col", "joint", "counts").

Value

a weightflow_estimation.

See Also

collect_estimates(), change_estimate(), panel_estimate(), level_estimate()

Examples

```
t1 <- subset(panel_ine, wave == 1 & disposition == "R")
t2 <- subset(panel_ine, wave == 2 & disposition == "R")
wb <- wave_bootstrap(
  list(T1 = weighting_spec(t1, base_weights = pw),
       T2 = weighting_spec(t2, base_weights = pw)),
  replicates = 100, strata = "stratum", psu = "psu", seed = 1, progress = FALSE)

# net change of the unemployment rate, by region
collect_estimates(wb |> step_domain(region) |>
  step_estimate(mean(unemployed), over = "change"))

# subpopulation: same change among the working-age population (rows masked, not dropped)
collect_estimates(wb |> step_filter(age >= 25 & age <= 54) |>
  step_estimate(mean(unemployed), over = "change"))
```

---

step_drop_ineligible *Drop ineligible (out-of-scope) units*

Description

Sets the weight of the units known to be outside the target population to zero, so they leave the cascade and take no part in any later step or in `collect_weights()`. Their weight is discarded, not redistributed: the weight total is meant to fall by exactly the mass they carried. Use it once eligibility has been resolved, immediately after `step_unknown_eligibility()`.

Usage

```
step_drop_ineligible(spec, ineligible, reason = NULL, id = NULL)
```

Arguments

`spec`	a weighting_spec.
`ineligible`	a 0/1 dummy column (1 = ineligible) or any logical condition (unquoted) that is TRUE for out-of-scope units.
`reason`	optional string naming why the units are out of scope (e.g. `"left the target population between waves"`). Recorded for the report narrative; it makes the panel distinction between a **between-wave exit** of the universe (the longitudinal population shrank – weight 0, no reweighting) and ordinary ineligibility explicit. Does not change the computation.
`id`	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived `"<class>_<k>"`.

Details

Apply it AFTER step_unknown_eligibility: ineligibles must be present and NOT flagged as unknown during that step, so they take part in the known-eligibility group and receive their share of the redistributed unknown weight. Their weight is then correctly discarded here (it represents the ineligible share of the unknown units, which are out of scope).

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
df <- transform(sample_survey,
                ineligible = as.integer(region == "West" & age > 90))
weighting_spec(df, base_weights = pw) |>
  step_drop_ineligible(ineligible = ineligible) |>
  prep()
```

---

step_model_calibration

Model calibration (Wu and Sitter 2001)

Description

Fits a working model for each study variable, predicts it over the whole population, and calibrates the weights so that the sample total of every prediction matches its population total, on top of the usual auxiliary totals – which may come from the population frame itself or from an external source (a census table, an administrative register). Reach for it when you hold, or can supply, those control totals and the outcome is well predicted by the auxiliaries: the predictions act as extra, highly relevant controls and buy precision that calibrating on `x` alone cannot.

Usage

```
step_model_calibration(
  spec,
  x_formula,
  models,
  population,
  x_totals = NULL,
  count = "Freq",
  by = NULL,
  cluster = NULL,
  equal_within_cluster = FALSE,
  calfun = c("linear", "logit", "raking"),
  bounds = NULL,
  maxit = 100L,
```

```
  tol = 1e-07,
  crossfit = NULL,
  crossfit_seed = NULL,
  id = NULL
)
```

Arguments

spec	a weighting_spec.
x_formula	formula of the consistency auxiliaries, e.g. ~ sex + region.
models	named list of models created with y_model(). The names label the prediction constraints.
population	population data.frame with the auxiliary and predictor columns (the y variables are not needed; they are predicted). May instead be a weighted reference survey wrapped with reference_sample(), in which case the totals are the design-weighted sums over that survey (estimated totals) rather than unweighted sums over a full frame. Always required: the model-assisted block predicts each y over every population unit, which cannot be done from aggregated totals.
x_totals	optional population totals for the consistency auxiliaries (x_formula), for when they come from an external source rather than from population (e.g. an official control total, a variable not present in the frame). Two shapes, the same as step_calibrate(method = "linear"): the tidy format, a named list matching the formula terms with a data frame (all categories + a counts column named by count) per factor and a single number per continuous total; or the classic model-matrix vector (intercept plus treatment contrasts). When NULL (default) the X totals are taken from population. When given, the X totals no longer require x_formula columns to exist in population (only in the sample), and population is used only for the model predictions.
count	name of the counts column in the tidy x_totals data frames. Only used when x_totals is given in the tidy (data-frame) format.
by	NULL (default), or a single column name: **partition** the sample by that domain and calibrate each domain on its own units, exactly as by does in step_calibrate(). For model calibration that carries two consequences at once, which is usually the reason for wanting it: every working model in models is fitted on the units of its own domain and never sees the other domains, and the x_formula totals are reproduced exactly **within** each domain rather than only nationally. population must carry the same domain column, and x_totals, if given, must be in the per-domain (tidy) form – a national named vector would be applied to every domain and the population counted once per domain. The rank ceiling then binds per domain, and harder: domain g must carry its own $q_A + K$ constraints on its n_g units, and with crossfit each of its folds must leave enough rows to fit the model coefficients. A domain too small for that is reported by name before anything is solved, because the symptom of a singular per-domain system is wild weights rather than an error. With linear working models, fitting by domain is the same mechanism as cross-fitting – the prediction columns leave the shared span – so the ceiling moves from $K \leq q - q_A$ to $K \leq Gq - q_A$ with G domains, and the two compose.

cluster	name of the cluster id column (e.g. "household"), for equal weights within the cluster.
equal_within_cluster	logical. If TRUE, integrative calibration: a single weight per cluster. Requires cluster and that the incoming weight be uniform within the cluster.
calfun	distance function for the calibration, as in step_calibrate(): "linear" (GREG, the default; closed form when unbounded), "raking" or "logit" (both solved by the Deville-Sarndal iteration). "logit" requires bounds.
bounds	optional numeric c(L, U) with L < 1 < U, bounding the calibration g-factor so the final weights stay in [L, U] times the incoming weight (same meaning and validation as in step_calibrate()). NULL (default) leaves the calibration unbounded. When set, the g-factors are found by the bounded Deville-Sarndal iteration, which keeps weights from turning negative or exploding; an infeasible range raises a non-convergence warning.
maxit, tol	iteration cap and convergence tolerance for the bounded / non-linear solver (ignored for unbounded calfun = "linear").
crossfit	integer or NULL. If given (K >= 2 folds), the outcome models are fitted by K-fold cross-fitting: the sample predictions are out-of-fold (each unit predicted by a model that did not see it), which avoids overfitting with flexible engines; the population total of the predictions uses the full model. Folds are formed by cluster when given. NULL (default) fits and predicts in-sample. For flexible learners cross-fitting is also what keeps the variance honest: same-sample residuals are shrunk by overfitting and can understate the variance even under recipe-aware replication (Dagdoug, Goga and Haziza 2023; Chernozhukov et al. 2018), so it is recommended whenever a model uses a non-glm engine.
crossfit_seed	integer or NULL. Seed for reproducible fold assignment.
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in collect_step_detail(); defaults to a derived "<class>_<k>".

Details

Requires COMPLETE auxiliary information: a data.frame population with the x_formula columns and the model predictors for the whole population (or a reference frame/census).

The predictions $\hat{y}_i$ enter as extra constraints, $\sum_{i \in s} w_i \hat{y}_i = \sum_{i \in U} \hat{y}_i$, solved together with the benchmark auxiliary totals $\mathbf{X}$. When the working model is linear this reduces to GREG; a non-linear learner adds efficiency through the prediction constraint while the totals $\mathbf{X}$ preserve design consistency even if the model is misspecified.

Value

The input weighting_spec with this step appended to its recipe. The step is recorded only; it is evaluated when prep() is called.

References

Wu, C. and Sitter, R. R. (2001). A model-calibration approach to using complete auxiliary information from survey data. *Journal of the American Statistical Association*, 96(453), 185-193. doi:10.1198/016214501750333054.

See Also

Other weighting steps: step_assert(), step_calibrate(), step_cre(), step_drop_ineligible(), step_nonresponse(), step_nr_sensitivity(), step_pseudoweight(), step_rescale(), step_round(), step_select_within(), step_subsample(), step_trim(), step_trim_calibrated(), step_trim_weights(), step_unknown_eligibility()

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_model_calibration(
    x_formula  = ~ sex + region,
    models     = list(income = y_model(income ~ age + sex, engine = "glm")),
    population = population) |>
  prep()

# with cross-fitting (out-of-fold predictions, avoids overfitting)
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_model_calibration(
    x_formula  = ~ sex + region,
    models     = list(income = y_model(income ~ age + sex, engine = "glm")),
    population = population, crossfit = 5, crossfit_seed = 1) |>
  prep()

# consistency totals from an external source (tidy format): a data frame per
# factor and a single number per continuous total. `population` is still used
# for the model predictions. Adjust for nonresponse first, since the outcome
# is only observed for respondents.
m_region <- as.data.frame(table(region = population$region))
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_model_calibration(
    x_formula  = ~ region + age,
    models     = list(income = y_model(income ~ age + sex, engine = "glm")),
    population = population,
    x_totals   = list(region = m_region, age = sum(population$age)),
    count      = "Freq") |>
  prep()

# equal weights within a household (integrative, Lemaitre-Dufour): one weight
# per cluster, so person and household estimates stay coherent. The final
# weights are constant within each cluster among its active members.
fit_hh <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_model_calibration(
    x_formula  = ~ sex + region,
    models     = list(income = y_model(income ~ age + sex, engine = "glm")),
    population  = population,
    cluster = "household_id", equal_within_cluster = TRUE) |>
  prep()
w <- fit_hh$final_weight
```

```
max(tapply(w[w > 0], sample_survey$household_id[w > 0],
           function(x) diff(range(x))))    # 0: one weight per household
```

`step_nonresponse` *Nonresponse adjustment*

Description

Inflates the weights of the eligible respondents so they also represent the eligible nonrespondents, under the assumption that response is ignorable given the information used. Three estimators are available – weighting classes, a response-propensity model (four engines, with optional cross-fitting), and calibration of the respondents to auxiliary totals – at the unit level or, through `cluster`, at a coarser level (e.g. the household).

Usage

```
step_nonresponse(
  spec,
  respondent,
  method = c("weighting_class", "propensity", "calibration"),
  by = NULL,
  formula = NULL,
  engine = c("logit", "tree", "forest", "boost"),
  weight_model = TRUE,
  num_classes = 5L,
  cluster = NULL,
  crossfit = NULL,
  crossfit_seed = NULL,
  totals = NULL,
  count = NULL,
  calfun = c("linear", "logit", "raking"),
  bounds = NULL,
  penalty = NULL,
  equal_within_cluster = FALSE,
  maxit = 50L,
  tol = 1e-06,
  id = NULL
)
```

Arguments

`spec`	a weighting_spec.
`respondent`	a 0/1 dummy column (1 = responded) or any logical condition (unquoted) TRUE for respondents. Eligible cases that are not respondents are treated as nonresponse.
`method`	"weighting_class" (cells), "propensity" (predictive model) or "calibration" (calibrate the respondents to auxiliary totals; two-phase / Sarndal-Lundstrom).

`by` character. Adjustment cells for method = "weighting_class".

`formula` predictor formula (right-hand side only), e.g. ~ age + region, used when method = "propensity".

`engine` engine to estimate the propensity when method = "propensity": "logit" (logistic regression, base R), "tree" (CART via package 'rpart'), "forest" (random forest via package 'ranger') or "boost" (gradient boosting via package 'xgboost'). 'rpart', 'ranger' and 'xgboost' are optional: only needed if you pick that engine. The flexible learners run with fixed default settings and their hyperparameters are not currently exposed: "tree" and "forest" use the 'rpart' and 'ranger' defaults, and "boost" uses xgboost with nrounds = 150, max_depth = 4 and eta = 0.1.

`weight_model` logical. Only for method = "propensity": whether to fit the response-propensity model with the incoming weights (`TRUE`, the default) or unweighted (`FALSE`). Fitting unweighted can reduce the variance of the propensity estimates when the weights are unrelated to response given the model covariates, at the cost of possible bias if they are (Little & Vartivarian 2003). The 1/p (or class) adjustment always uses the design weights; only the model fit is affected.

`num_classes` integer or NULL. Controls how propensities are used: an integer forms that many propensity classes (cell adjustment within each class); NULL applies the direct factor 1/p to each unit. When the fitted propensities are (nearly) constant the requested quantile classes cannot be formed; rather than error, or fabricate classes by jittering the propensities (which is not reproducible and invents structure that is not there), all units are placed in a single adjustment class and a quality alert is raised – the statistically correct outcome, since equal propensities give nothing to differentiate.

`cluster` character or NULL. If given, the adjustment is done at the cluster level for whole-cluster nonresponse: each cluster counts once with its (uniform) weight; in "weighting_class" the redistribution is between responding and nonresponding clusters within the cells, and in "propensity" the model is fitted with one row per cluster (cluster auxiliaries), predicting the cluster's response. The resulting factor is assigned to every member; nonresponding clusters go to zero. As always, only active units (weight > 0) take part, so units already dropped (unknown eligibility, ineligible) are excluded automatically. For `method = "calibration"`, `cluster` is used together with `equal_within_cluster = TRUE` for integrative (one weight per cluster) calibration.

The cluster need not be a household: it is any grouping whose members share a fate and a weight – a dwelling, an area segment, or a whole primary sampling unit (an entire PSU inaccessible, then redistributed within its stratum). A methodological consequence to keep in mind: a cluster-level adjustment preserves the *mass of clusters* in each cell (the cluster weight is the mean of its members, in the sense of Valliant et al. 2018), and the factor is uniform within the cell; it does **not**, by construction, preserve the mass of the underlying units (persons). That is the job of the later calibration, whose margins bring the person totals back exactly.

`crossfit` integer or NULL. If given (number of folds K >= 2), the propensity is estimated by K-fold cross-fitting: for each fold the model is trained on the other folds and used to predict the held-out fold, so each unit's propensity comes from a model

that did not see it. This avoids the overfitting that flexible engines (forest, boost) can produce, which would otherwise inflate the weights. Folds are formed by `cluster` when given (so correlated units stay together). NULL (default) fits and predicts in-sample. For flexible learners it also keeps the design-based variance honest: same-sample predictions can understate the variance even under recipe-aware replication (Dagdoug, Goga and Haziza 2023), so cross-fitting is recommended whenever `engine` is not "logit".

`crossfit_seed` integer or NULL. Seed for reproducible fold assignment when `crossfit` is used.

`totals` (method = "calibration") calibration targets. NULL (default) calibrates the respondents to the R+NR design-weighted totals of `formula` at that stage (the two-phase / sample-level case; Sarndal & Lundstrom 2005); a named vector or a tidy `totals`/`count` input (as in `step_calibrate()`) calibrates to population totals instead.

`count` (method = "calibration", tidy `totals`) string naming the counts column of the totals data frame(s).

`calfun` (method = "calibration") distance function for the calibration factor: "linear", "raking" or "logit", as in `step_calibrate(method = "linear")`.

`bounds` (method = "calibration") numeric c(L, U) with L < 1 < U. Bounds on the calibration factor, to keep the nonresponse factors positive.

`penalty` (method = "calibration", unbounded) NULL or positive cost(s) for ridge (penalized) calibration.

`equal_within_cluster`
(method = "calibration") logical. If TRUE, integrative (Lemaitre-Dufour) nonresponse calibration: the responding members of a household (`cluster`) share a single calibration factor, so the adjustment keeps the weights constant within household. Requires `cluster`. FALSE (default) calibrates each responding unit on its own.

`maxit`, `tol` (method = "calibration") convergence control for the bounded or exponential-distance calibration solver.

`id` optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived "<class>_<k>".

Details

The three estimators are the same operation with a different inflation factor. Weighting classes inflate the responding weight to the cell total, $f_c = \sum_{i \in c} w_i / \sum_{i \in c} r_i w_i$ (with $r_i = 1$ if i responds), applied as $w_i \leftarrow f_c w_i$. A response-propensity model instead adjusts unit by unit, $w_i \leftarrow w_i / \hat{\phi}_i$, with $\hat{\phi}_i$ the estimated response propensity. Calibration of the respondents solves for a factor v_i that makes the respondents reproduce a reference total (the two-phase / Sarndal-Lundstrom approach).

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: step_assert(), step_calibrate(), step_cre(), step_drop_ineligible(), step_model_calibration(), step_nr_sensitivity(), step_pseudoweight(), step_rescale(), step_round(), step_select_within(), step_subsample(), step_trim(), step_trim_calibrated(), step_trim_weights(), step_unknown_eligibility()

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class",
                   by = "region")

# household-level nonresponse (whole household responds or not)
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class",
                   by = "region", cluster = "household_id") |>
  prep()
# propensity with cross-fitting (out-of-sample, avoids overfitting)
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "propensity",
                   formula = ~ region + sex, engine = "logit",
                   num_classes = 5, crossfit = 5, crossfit_seed = 1) |>
  prep()

# gradient boosting engine (requires the 'xgboost' package)
if (requireNamespace("xgboost", quietly = TRUE)) {
  weighting_spec(sample_survey, base_weights = pw) |>
    step_nonresponse(respondent = responded, method = "propensity",
                     formula = ~ region + sex + age, engine = "boost",
                     num_classes = 5, crossfit = 5) |>
    prep()
}

# nonresponse by calibration (two-phase): calibrate the respondents to the
# R+NR design-weighted totals of the auxiliaries at that stage, so their
# estimates reproduce the pre-nonresponse ones (Sarndal & Lundstrom 2005)
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "calibration",
                   formula = ~ region + sex) |>
  prep()
```

`step_nr_sensitivity` *Sensitivity of a mean to nonignorable nonresponse or selection*

Description

A diagnostic step (it does not change any weight) that gauges how much the weighted mean of a study variable could move if response, or participation in a non-probability sample, depended on the outcome itself beyond the observed auxiliaries. It implements the proxy pattern-mixture model of

Andridge and Little (2011): the auxiliaries are reduced to a single proxy (the respondent regression prediction of `y`), and a single sensitivity parameter `phi` in `[0, 1]` moves the mechanism from ignorable given the proxy (`phi = 0`, MAR) to depending only on the outcome (`phi = 1`). Evaluated over a grid of `phi`, the adjusted means form an *ignorance interval* to read alongside the sampling confidence interval; see `nr_sensitivity()` and the report block.

Usage

```
step_nr_sensitivity(
  spec,
  y,
  formula,
  respondent = NULL,
  eligible = NULL,
  phi = c(0, 0.25, 0.5, 0.75, 1),
  id = NULL
)
```

Arguments

`spec`	a `weighting_spec`.
`y`	the study variable (bare column name), observed for respondents and `NA` for nonrespondents.
`formula`	one-sided formula of the auxiliaries for the proxy, observed for all units, e.g. `~ region + sex + age`.
`respondent`	optional response/participation indicator (bare column or condition). Defaults to `!is.na(y)`.
`eligible`	optional in-scope indicator (bare column or condition), the mirror of the argument in `step_nonresponse()`. Out-of-scope (ineligible) units are neither respondents nor nonrespondents and must be excluded, or they would be counted as nonrespondents and pull the estimate toward their proxy mean. Give it in any household survey that has ineligible units. Default `NULL` treats every active unit as in scope.
`phi`	the sensitivity grid, values in `[0, 1]`; `0` (MAR) is always added. Little et al. (2020) suggest `0.5` as a central value; above `0.5` the implied mechanism is often unrealistically strong.
`id`	optional stable step id.

Details

The adjusted mean at sensitivity ϕ is

$$\mu(\phi) = \bar{y}_r + (1 - \pi)\,\frac{s_{yr}}{s_{xr}}\,m(\phi)\,(\bar{x}_{nr} - \bar{x}_r),$$

with slope $m(\phi) = \frac{(1-\phi)\rho+\phi}{(1-\phi)+\phi\rho}$, so that $m(0) = \rho$ (ignorable given the proxy) and $m(1) = 1/\rho$.

The proxy correlation `rho` (the multiple correlation of `y` on the auxiliaries among respondents) sets how informative the auxiliaries are: a weak proxy widens the ignorance interval (at `phi = 1` the

slope is 1/rho). The step reads the base design weights, so place it anywhere in the recipe; it needs the nonrespondents still present (a study variable that is NA for them, or an explicit respondent indicator).

Value

the input weighting_spec with this diagnostic step appended.

References

Andridge, R. R. and Little, R. J. A. (2011). Proxy pattern-mixture analysis for survey nonresponse. Journal of Official Statistics 27(2), 153-180.

See Also

nr_sensitivity(), step_assert()

Other weighting steps: step_assert(), step_calibrate(), step_cre(), step_drop_ineligible(), step_model_calibration(), step_nonresponse(), step_pseudoweight(), step_rescale(), step_round(), step_select_within(), step_subsample(), step_trim(), step_trim_calibrated(), step_trim_weights(), step_unknown_eligibility()

---

step_panel_overlap *Adjust base weights by the panel-selection probability (CEPAL ch. XVI)*

Description

Divides each incoming weight by Pr(panel selection) – the probability that a rotation panel belongs to the combined-wave sample – turning the design weight of the reference wave into the panel base weight of the longitudinal sample (d_base = d1 / Pr). This is the first step of the Verma-Betti-Ghellini longitudinal-weight sequence: combining k waves in a design that keeps g of its G rotation groups in all of them gives Pr = g / G, so the factor is its reciprocal (e.g. 4/3 when 3 of 4 groups persist, 4 when only 1 does).

Usage

```
step_panel_overlap(spec, prob, id = NULL)
```

Arguments

spec	a weighting_spec.
prob	a panel-selection probability in (0, 1]: an unquoted per-unit column/expression, or a single constant (e.g. panel_pr(pd, c("T1","T2"))).
id	optional string identifier for the step, shown in the recipe print-out and usable in collect_step_detail().

Details

prob is a probability in (0, 1]: either a per-unit column (or expression) or a single constant applied to every active unit. When the data has a rotation_group, get the constant from panel_pr(): prob = panel_pr(pd, c("T1", "T2")). When it does not (e.g. the Chilean ENE), derive the probability yourself (from the design fraction or the cluster) and pass it here. Run this on the longitudinal sample (the intersection of the combined waves), before the attrition and calibration steps.

Value

The input weighting_spec with this step appended to its recipe.

See Also

panel_pr(), panel_design(), step_nonresponse(), step_calibrate()

Examples

```
# wide longitudinal file of the units in sample in both waves
wide <- panel_merge(
  list(T1 = subset(panel_ine, wave == 1), T2 = subset(panel_ine, wave == 2)),
  by = c("household_id", "person_no"), require = "all")

# rotation group known -> derive Pr(panel selection) from the panel design
pd <- panel_design(panel_ine, unit = c("household_id", "person_no"), wave = "wave",
                   rotation_group = "rotation_group")
weighting_spec(wide, base_weights = pw_T1) |>
  step_panel_overlap(prob = panel_pr(pd, c("1", "2"))) |> prep()

# no rotation group (e.g. Chile ENE) -> pass the probability directly
weighting_spec(wide, base_weights = pw_T1) |>
  step_panel_overlap(prob = 5 / 6) |> prep()
```

---

step_pseudoweight *Pseudo-weights for a non-probability sample against a reference*

Description

For a non-probability sample (opt-in panel, volunteer or river sample) with no design weights, step_pseudoweight() estimates each unit's *participation propensity* $\hat{p}$ against a probability reference_sample() and assigns the pseudo-weight $(1 - \hat{p})/\hat{p}$ (the participation odds; the adjusted logistic propensity of Wang, Valliant and Li 2021, after Elliott and Valliant 2017), which inflates each unit to the population. Their sum *estimates* the reference's estimated population size – it is unbiased for it, not equal to it, and varies from sample to sample like any other estimator. It stacks the non-probability sample and the reference internally (the participation indicator and the two samples' weights are built for you), fits the propensity, and returns the pseudo-weight on the non-probability units only; the reference is used to train the model and then dropped.

Usage

```
step_pseudoweight(
  spec,
  reference,
  formula,
  engine = c("logit", "tree", "forest", "boost"),
  num_classes = NULL,
  by = NULL,
  crossfit = NULL,
  crossfit_seed = NULL,
  id = NULL
)
```

Arguments

spec	a non-probability `weighting_spec`.
reference	a `reference_sample()` (the probability reference with its design weights). Pass the reference's replicate weights through `reference_sample(replicates = )` to propagate its sampling variance through the recipe-aware bootstrap: each replicate refits the propensity from the paired reference replicate. Without them the reference is treated as fixed, so the bootstrap reflects only the variability of the non-probability sample (which it resamples as a with-replacement sample of units, a slightly conservative approximation when that sample is a large fraction of the population).
formula	one-sided formula of the covariates shared by both samples, e.g. `~ sex + age + region`.
engine	propensity learner: `"logit"` (default), `"tree"`, `"forest"` or `"boost"`.
num_classes	NULL (default, direct `1/pi`) or an integer: group the fitted propensities into that many quantile classes and use the class-average pseudo-weight, which is more robust to a misspecified model.
by	NULL (default), or a single column name: fit **one participation model per domain**, on that domain's units of both samples and on nothing else. The `reference` must carry the same domain column, and is split by it, so the pseudo-weights of a domain sum to that domain's reference total rather than to the national one – which is what lets a following domain-partitioned calibration start from a coherent input weight. Use it when the participation mechanism differs across domains (a covariate whose effect changes sign by region is the clear case), where a single *additive* national model misses the domain totals in both directions – measured on three domains with the slope changing sign: `~ dom + x` landed at 4,258 / 3,530 / 4,803 against a truth of 4,022 / 4,033 / 3,945. For `engine = "logit"` this is the fully interacted national model, `~ dom * (...)`, and gives the same weights to numerical precision – not a different estimator, but the practicable way to fit it: the interactions are never written out, a domain without enough units on either side is named instead of quietly degrading the whole fit, and the tree, forest and boost engines have no interaction syntax at all, so there segmenting is the only route. Each domain needs enough units on **each** side to fit its own model; those that do not are reported by name.

`crossfit`, `crossfit_seed`
: optional K-fold cross-fitting of the propensity (recommended for the flexible learners), and its seed.

`id`
: optional stable step id.

Details

The recipe must be a non-probability spec: `weighting_spec(..., nonprob = TRUE)`. This step is the inverse-propensity (IPW) route; you can instead, or additionally, calibrate to a `reference_sample()` with `step_calibrate()` / `step_model_calibration()` (mass imputation / model-based), and combining both gives the doubly robust estimator.

Value

the input `weighting_spec` with this step appended.

References

Wang, L., Valliant, R. and Li, Y. (2021). Adjusted logistic propensity weighting methods for population inference using nonprobability volunteer-based epidemiologic cohorts. Statistics in Medicine 40(24), 5237-5250. doi:10.1002/sim.9122

Elliott, M. R. and Valliant, R. (2017). Inference for non-probability samples. Statistical Science 32(2), 249-264.

See Also

`reference_sample()`, `step_calibrate()`, `step_model_calibration()`

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
set.seed(1)
N   <- nrow(population)
# a biased volunteer sample (men over-participate) and a probability reference
vol <- population[rbinom(N, 1, plogis(-2 + 0.9 * (population$sex == "M"))) == 1,
                  c("region", "sex", "income")]
ref <- population[sample(N, 600), c("region", "sex")]
ref$d <- N / 600                                  # its design weights
fit <- weighting_spec(vol, base_weights = NULL, nonprob = TRUE) |>
  step_pseudoweight(reference = reference_sample(ref, "d"),
                    formula = ~ region + sex, engine = "logit") |>
  prep()
# the pseudo-weighted mean corrects the volunteer bias
c(naive = mean(vol$income),
  pseudo = weighted.mean(vol$income, fit$final_weight),
  truth = mean(population$income))
```

step_rescale *Rescale the weights to a fixed sum*

Description

Multiplies the active weights by a single constant so that they add up to a chosen total: either the number of active units (mean weight 1) or an arbitrary number. Use it as a presentation step, when the analysis wants normalized weights rather than population-scale ones.

Usage

```
step_rescale(spec, to = c("n", "total"), total = NULL, by = NULL, id = NULL)
```

Arguments

spec	a weighting_spec.
to	"n" (weights sum to the number of active units, i.e. mean weight 1) or "total" (weights sum to total).
total	numeric. Target sum when to = "total".
by	character. Rescale within these groups (optional). With to = "n", each group sums to its own active count.
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in collect_step_detail(); defaults to a derived "<class>_<k>".

Value

The input weighting_spec with this step appended to its recipe. The step is recorded only; it is evaluated when prep() is called.

See Also

Other weighting steps: step_assert(), step_calibrate(), step_cre(), step_drop_ineligible(), step_model_calibration(), step_nonresponse(), step_nr_sensitivity(), step_pseudoweight(), step_round(), step_select_within(), step_subsample(), step_trim(), step_trim_calibrated(), step_trim_weights(), step_unknown_eligibility()

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_rescale(to = "n") |> prep()
```

step_round *Round the final weights*

Description

Rounds the weights to a given number of decimals, either unit by unit (`"nearest"`), with the largest-remainder method (`"preserve_total"`), which keeps the weighted total exactly, or with the cube method (`"balanced"`), which keeps the calibrated totals – by domain, not only the grand total – as close as the integer grid allows. For `"balanced"`, `formula` balances on the calibration design and `by` on crossed cells; they are different problems, and the difference is spelled out under those arguments. Typically the last step of a recipe, after calibration, when the weights have to be delivered as integers or with a fixed number of decimals.

Usage

```
step_round(
  spec,
  digits = 0L,
  method = c("nearest", "preserve_total", "balanced"),
  by = NULL,
  formula = NULL,
  id = NULL
)
```

Arguments

`spec`	a weighting_spec.
`digits`	integer. Decimals to keep (0 = integers).
`method`	one of `"nearest"` (simple rounding), `"preserve_total"` (largest remainder; keeps the grand total exactly) or `"balanced"` (cube method; keeps the totals of the domains named in `by` as close as the grid allows). Note: `"preserve_total"` and `"balanced"` can break equality of weights within a cluster; if you need integer and equal weights per household, use `"nearest"`.
`by`	for `method = "balanced"` only, and an alternative to `formula`: a character vector of variables whose **crossed** cell totals must be preserved, e.g. `by = c("dam", "stratum")`. The balancing matrix is then one indicator per non-empty cell, so every cell total – and hence every margin, and the grand total – is reproduced up to about one unit's worth. This is the right choice after a post-stratification, where the cells *are* the calibration. It is **stricter** than `formula`, not equivalent: preserving every cell implies preserving the margins, but it also imposes constraints the calibration never asked for, and with many sparse cells the rounding cannot meet them all. Note too that with cell indicators each unit loads on a single column, so the problem separates into one independent cell total at a time and the cube method has no overlap to exploit; the overlap of `model.matrix()` columns is what the method is for.

formula	for method = "balanced" only: a one-sided formula, e.g. ~ dam + stratum. The balancing matrix is model.matrix(formula, data) over the active units, so the rounding reproduces **exactly the totals a calibration on that same formula reproduces**, and nothing else. This is the argument to use after step_calibrate(formula =): pass the same formula, or pass neither formula nor by and the step takes it from the last calibration step in the recipe.
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in collect_step_detail(); defaults to a derived "<class>_<k>".

Details

The "balanced" method implements balanced rounding by the cube method (Deville and Tille 2004; ECLAC/CEPAL household-survey methodology, chapter 9, section F.2) natively, with no external sampling dependency. It is randomized: call set.seed() before prep() for a reproducible result.

"preserve_total" is randomized too, but only where it has to be. Ties in the fractional part are the rule rather than the exception – a self-weighting design, weights that land on .5, calibrated weights on a grid – and they are broken at random, so the extra unit is allocated without regard to the order of the file. Deciding ties by row order instead moves mass systematically towards whatever the file is sorted by, usually region, while the grand total (the one thing the method promises) stays exactly right and hides it. Weights whose fractional parts are distinct are rounded exactly as before.

Value

The input weighting_spec with this step appended to its recipe. The step is recorded only; it is evaluated when prep() is called.

References

Deville J-C, Tille Y (2004). Efficient balanced sampling: the cube method. *Biometrika* 91(4):893-912.

See Also

Other weighting steps: step_assert(), step_calibrate(), step_cre(), step_drop_ineligible(), step_model_calibration(), step_nonresponse(), step_nr_sensitivity(), step_pseudoweight(), step_rescale(), step_select_within(), step_subsample(), step_trim(), step_trim_calibrated(), step_trim_weights(), step_unknown_eligibility()

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_round(digits = 0) |> prep()
```

`step_select_within` *Within-cluster selection adjustment*

Description

Undoes one stage of subsampling inside a cluster: when only some of the eligible units of a household (or dwelling, or area segment) were selected, the selected ones must represent the whole cluster, so their weight is multiplied by the inverse of the within-cluster selection probability. Apply it after the cluster-level eligibility and nonresponse steps and before the person-level nonresponse step.

Usage

```
step_select_within(
  spec,
  prob = NULL,
  n_eligible = NULL,
  n_selected = NULL,
  id = NULL
)
```

Arguments

`spec`	a weighting_spec.
`prob`	unquoted column with the within-household selection probability of the selected person (need not be 1/n_eligible). The weight is multiplied by 1/prob.
`n_eligible`	unquoted column with the number of eligible persons in the household, for simple random selection within the household. When a single person is selected (the default), the weight is multiplied by n_eligible (equivalent to prob = 1/n_eligible).
`n_selected`	optional number of persons selected per household under simple random selection, when more than one person is subsampled. Either a single number (same subsample size in every household) or an unquoted column (subsample size varying by household). The weight is multiplied by n_eligible / n_selected (equivalent to prob = n_selected/n_eligible). Defaults to 1. Only used together with `n_eligible`.
`id`	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived "<class>_<k>".

Details

Despite the name, the cluster need not be a household and the unit need not be a person: the step is the generic within-cluster subsampling adjustment. In a multi-stage design it can appear more than once – e.g. dwellings selected within sampled area segments, then persons selected within dwellings – each occurrence undoing one stage of subsampling.

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
# simple random selection of one eligible person per household
df <- transform(sample_survey,
                n_elig = ave(person_id, household_id, FUN = length))
weighting_spec(df, base_weights = pw) |>
  step_select_within(n_eligible = n_elig)

# simple random selection of two eligible persons per household
weighting_spec(df, base_weights = pw) |>
  step_select_within(n_eligible = n_elig, n_selected = 2)
```

---

step_subsample *Second-phase subsampling (two-phase sampling)*

Description

Undoes a second phase of sampling: when a subsample of the first-phase units was drawn for a more expensive follow-up (measuring an outcome, a longer questionnaire), the subsampled units must represent the whole first-phase sample. Their weight is multiplied by the inverse of the phase-2 selection probability, and the not-subsampled units leave the cascade (weight 0).

Usage

```
step_subsample(spec, selected, prob, psu, id = NULL)
```

Arguments

`spec`	a weighting_spec.
`selected`	a 0/1 dummy column (1 = selected in phase 2) or any logical condition (unquoted) TRUE for the subsampled units. Units that are not selected leave the cascade (weight 0).
`prob`	unquoted column with the phase-2 selection probability π_2 of the selected units. The weight is multiplied by 1/prob. Must be in (0, 1] for every selected unit and constant within each phase-2 sampling unit.

psu — character. The phase-2 sampling unit column (e.g. the household id at which the subsample was drawn). The two-phase resampling factor is generated at this level and shared by the members of the unit.

id — optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived `"<class>_<k>"`.

Details

The step also *records the phase-2 design* (the selection probability and the phase-2 sampling unit) so that `bootstrap_weights()` can reproduce the two-phase variance $V = V_1 + V_2$: the phase-1 sampling variance plus the expected conditional variance of the phase-2 subsample. A single-phase bootstrap would only capture V_1 and undercover.

The coupling is additive, not multiplicative: the per-unit resampling factor has variance $(1-f_1)\pi_2 + (1-\pi_2)$, the sum of the phase-1 component $(1-f_1)\pi_2$ (seen through the subsample) and the phase-2 conditional component $1-\pi_2$. A naive product of two factors adds a spurious interaction term and is too wide. In practice the factor is drawn once per phase-2 sampling unit from a strictly positive Gamma of that mean and variance, so every replicate weight stays positive (a downstream propensity/GLM step re-runs cleanly). See `vignette("two-phase-sampling")` for the methodology and its Monte Carlo validation.

The second phase is modelled as Poisson (independent / Bernoulli) selection of the sampling unit nested in the first phase (e.g. households subsampled from a first-phase household sample). This is the general-purpose choice: a Poisson second phase is conservative for, and closely approximates, the without- replacement and stratified subsampling schemes used in practice when the phase-2 sampling fraction is small – which is the usual case, since a costly follow-up subsamples only a fraction of the first phase. The phase-1 sampling fraction f_1 is taken from the `fpc` argument of `bootstrap_weights()` and defaults to 0 (negligible, the usual case in household surveys), which reduces the coupling to a single independent per-unit factor of variance 1 – a Gamma of variance 1, i.e. an Exponential(1) (the Bayesian-bootstrap multiplier): valid and strictly positive, but right-skewed, which is part of why replicate-to-replicate variance estimates have heavier tails.

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

`bootstrap_weights()` for the two-phase variance; `step_select_within()` for within-cluster subsampling that is not a separate sampling phase.

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
# households subsampled for a follow-up module, selected with prob p2
df <- transform(sample_survey,
```

```
                in_phase2 = as.integer(runif(nrow(sample_survey)) < 0.3),
                p2 = 0.3)
weighting_spec(df, base_weights = pw) |>
  step_subsample(selected = in_phase2, prob = p2, psu = "household_id")
```

`step_trim` *Trim extreme weights against a ratio*

Description

Caps the current weights at a multiple of a reference (each unit's base weight, the group median, or an absolute value) and, by default, redistributes the removed mass among the untrimmed units so the weighted total survives the trim. With by, the reference and the cap are computed separately within each subgroup. An optional step that can appear anywhere in the recipe, more than once.

Usage

```
step_trim(
  spec,
  max_ratio,
  min_ratio = NULL,
  reference = c("base", "median", "value"),
  redistribute = TRUE,
  by = NULL,
  maxit = 50L,
  id = NULL
)
```

Arguments

`spec`	a weighting_spec.
`max_ratio`	number. Upper cap. Its meaning depends on `reference`. E.g. with reference = "base" and max_ratio = 4, no weight may exceed 4 times its design weight. Must be greater than 1 for reference = "base"/"median" (a multiplier) and greater than 0 for reference = "value" (an absolute weight).
`min_ratio`	number or NULL. Lower floor (same units as max_ratio); if supplied, must be greater than 0 and below `max_ratio`.
`reference`	"base" (multiple of each unit's base weight), "median" (multiple of the median of current weights) or "value" (absolute weight value).
`redistribute`	logical. If TRUE, redistributes the trimmed excess among the uncapped weights to preserve the total (iterating). If you calibrate afterwards you can use FALSE: calibration restores the totals.
`by`	character. Groups within which to redistribute (optional).
`maxit`	integer. Maximum cap+redistribution iterations.
`id`	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived "<class>_<k>".

Details

There is no standard threshold: `max_ratio` is an analyst decision, a bias-variance trade-off. Use Kish's design effect (see summary) to judge whether trimming is worth it.

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim_calibrated()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_trim(max_ratio = 3, reference = "base")
```

---

step_trim_calibrated *Trimmed calibration (range-restricted, totals-preserving)*

Description

Pulls already-calibrated weights into an absolute interval `[lower, upper]` without breaking the calibration: instead of capping and redistributing, it re-solves a bounded calibration whose targets are the totals the incoming weights already reproduce, optionally with its own band per subgroup through `by`. It is the only one of the three trimming steps that leaves the calibration totals intact.

Usage

```
step_trim_calibrated(
  spec,
  formula,
  lower = NULL,
  upper = NULL,
  calfun = c("linear", "raking"),
  by = NULL,
  cluster = NULL,
  equal_within_cluster = FALSE,
  maxit = 100L,
  tol = 1e-07,
  id = NULL
)
```

Arguments

spec	a weighting_spec.
formula	the auxiliaries whose calibration totals must be preserved (right-hand side only), e.g. ~ region + age_group. Usually the same formula used in the preceding step_calibrate() (or the x_formula of the preceding step_model_calibration(), whose model-prediction totals are then preserved as well).
lower, upper	numeric. Absolute bounds on the trimmed weight. At least one must be supplied; the other defaults to no bound. For positive variance, use a positive lower. Each may be a single number (the same bound for every unit) or, together with by, a named vector of bounds per subgroup (names = the by group levels), for differentiated trimming.
calfun	distance function: "linear" (default; the range-restricted Euclidean distance) or "raking" (the multiplicative distance, which keeps the adjustment factors positive).
by	character or NULL. Subgroup column for differentiated bounds: with a named-vector lower/upper, each subgroup is trimmed to its own bounds while the preserved totals of formula stay global. NULL (default) uses the same bounds for all units.
cluster	character or NULL. Cluster (e.g. household) id column, for integrative trimming (with equal_within_cluster = TRUE).
equal_within_cluster	logical. If TRUE, integrative trimming: one trimming factor per cluster, so weights stay constant within household. The incoming weights must already be constant within cluster (e.g. from step_calibrate(equal_within_cluster = TRUE)); the absolute bound then applies to that common household weight. Requires cluster. FALSE (default) trims each unit on its own.
maxit	integer. Maximum iterations for the bounded solver.
tol	numeric. Convergence tolerance for the bounded solver.
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in collect_step_detail(); defaults to a derived "<class>_<k>".

Details

The absolute-weight bound is imposed as a per-unit factor bound w_new / w in [lower/w, upper/w] on top of the incoming weights, using a bounded (range-restricted) calibration with the truncated Deville-Sarndal distances: the range-restricted Euclidean distance (calfun = "linear", the default) or the multiplicative one (calfun = "raking"). Weights inside the range that are not needed to restore the totals stay put; the out-of-range ones saturate at their bound and the rest move as little as possible. If the range is too tight to preserve every total, the totals that cannot be met are relaxed and a warning is raised.

This step is meant to run **after** a step_calibrate() or a step_model_calibration(): it acts on the active incoming weights (including any negative weights an unbounded linear calibration produced, which it can bring back into [lower, upper]) and leaves dropped units (weight 0) alone. After a step_model_calibration() it preserves both the known-margin totals and the model-prediction totals: that step saves its prediction columns, which this step appends to formula's

design so the trimmed weights keep every total the model calibration reproduced (pass the same `x_formula` as `formula`).

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

References

Deville, J.-C. and Sarndal, C.-E. (1992). Calibration estimators in survey sampling. Journal of the American Statistical Association, 87, 376-382. doi:10.2307/2290268. The totals-preserving trimming solves a bounded (range-restricted) calibration with the truncated distances introduced there. Folsom, R. E. and Singh, A. C. (2000). The generalized exponential model for sampling weight calibration for extreme values, nonresponse and poststratification. Proceedings of the ASA Survey Research Methods Section, 598-603, formalises the same range-restricted (generalized exponential) family.

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_weights()`, `step_unknown_eligibility()`

Examples

```
# calibrate, then trim the calibrated weights into [5.5, 13.5] without breaking
# the region/sex totals (the calibrated weights of sample_survey live in ~[5.4, 14])
weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 margins = list(region = c(table(population$region)),
                                sex    = c(table(population$sex)))) |>
  step_trim_calibrated(~ region + sex, lower = 5.5, upper = 13.5) |>
  prep()
```

---

step_trim_weights *Automatic weight trimming to an absolute band*

Description

Caps the weights into an absolute interval `[lower, upper]` and hands the removed mass back to the units that were not capped, so the weighted total is preserved. This is the step to use when you have **not calibrated yet** (or will calibrate afterwards) and you want the cutoff chosen from the data rather than argued for: with `upper = NULL` it picks one by the Tukey far-out fence or by Potter's MSE rule.

Usage

```
step_trim_weights(
  spec,
  lower = 1,
  upper = NULL,
  method = c("tukey", "potter"),
  redistribute = c("proportional", "uniform"),
  strict = TRUE,
  maxit = 50L,
  kappa = 1,
  id = NULL
)
```

Arguments

spec	a weighting_spec.
lower	numeric. Lower floor (default 1: no weight below 1).
upper	numeric or NULL. Upper cap. If NULL, the cap is chosen automatically by method.
method	rule for the automatic cap when upper = NULL: "tukey" (default, Q3 + 3*IQR far-out fence) or "potter" (Potter's MSE-optimal cutoff, which over a grid of candidate cutoffs minimizes an estimate of bias^2 + variance and so balances the bias of trimming against the variance from extreme weights). Ignored when upper is supplied. See Details for what the Potter criterion assumes.
redistribute	how the trimmed mass is shared among the untrimmed units: "proportional" (default; in proportion to their weights, preserving relative sizes) or "uniform" (an equal amount to each untrimmed unit, and units already trimmed are not reused, exactly reproducing survey::trimWeights()).
strict	logical. If TRUE (default), iterate cap+redistribution until no weight is outside [lower, upper] (like survey's strict = TRUE). If FALSE, a single pass (redistribution may push some weights slightly past the cap).
maxit	integer. Maximum iterations when strict = TRUE.
kappa	numeric, method = "potter" only: the relative price of bias against variance in the criterion minimized, kappa * bias^2 + variance. The default 1 is Potter's own weighting; above 1 the cutoff moves up (trim less, keep the bias down), below 1 it moves down (trim more).
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in collect_step_detail(); defaults to a derived "<class>_<k>".

Details

Two things are worth knowing about method = "potter" before reading its cutoff as optimal, because both are assumptions rather than results.

The criterion is the mean squared error of a **total**, so its two terms are deliberately on different orders: the bias of capping is the trimmed mass, which grows like the sample size, and the variance term is the sum of squared remaining weights, which also grows like the sample size, so bias

squared grows like its square. The cutoff therefore **rises with the sample size for the same weight distribution** – replicating a 300-unit sample to 30,000 moved it from 126 to 183 in one test, capping a third as many units. That is the correct behaviour for a total (the relative bias stays put while the relative variance shrinks, so trimming buys less), not a defect, but it does mean the rule is not a property of the weight distribution alone. The criterion is invariant to the scale of the weights.

The bias it charges is the bias of capping **without** redistribution, while this step always redistributes: the weighted total is preserved exactly, so the real bias is only the difference between the study variable's mean among the capped units and among the units receiving their mass. The criterion therefore overstates the bias and, other things equal, caps less than the MSE it is named after would. `kappa` is the handle: set it below 1 to buy back that conservatism.

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_unknown_eligibility()`

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_trim_weights(lower = 1, strict = TRUE) |> prep()

# Potter MSE-optimal cutoff chosen from the data
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_trim_weights(method = "potter") |> prep()
```

`step_unknown_eligibility`

Unknown-eligibility adjustment

Description

Redistributes the weight of the cases whose eligibility was never resolved onto the resolved cases of the same adjustment cell, so the resolved units stand in for the unresolved share of the frame. Reach for it as the first step of the cascade, while the known-ineligible units are still in the data.

Usage

```
step_unknown_eligibility(spec, unknown, by = NULL, cluster = NULL, id = NULL)
```

Arguments

spec	a weighting_spec.
unknown	a 0/1 dummy column (1 = eligibility unknown) or any logical condition (unquoted) that is TRUE for unknown-eligibility cases. Evaluated on the data.
by	character. Variables defining the adjustment cells (optional).
cluster	character. Cluster (e.g. household) id column. If given, the redistribution is done at the cluster level: each cluster counts once with its (uniform) weight, the weight of unknown-eligibility clusters is redistributed among the known ones, and the adjusted weight is assigned to every member. Use this when unknown-eligibility units have no roster (one row per address) while resolved units are expanded by person.
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in `collect_step_detail()`; defaults to a derived `"<class>_<k>"`.

Details

Within each cell c the resolved cases are scaled up to also carry the unresolved ones,

$$w_i^{\text{out}} = w_i \, \frac{\sum_{j \in c} w_j}{\sum_{j \in c,\, \text{resolved}} w_j},$$

with every weight on the right the weight *entering* the step, so the ratio is one number per cell, the cell total is conserved exactly, and the result does not depend on the order in which units are updated.

Value

The input `weighting_spec` with this step appended to its recipe. The step is recorded only; it is evaluated when `prep()` is called.

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `step_cre()`, `step_drop_ineligible()`, `step_model_calibration()`, `step_nonresponse()`, `step_nr_sensitivity()`, `step_pseudoweight()`, `step_rescale()`, `step_round()`, `step_select_within()`, `step_subsample()`, `step_trim()`, `step_trim_calibrated()`, `step_trim_weights()`

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_unknown_eligibility(unknown = unknown_elig, by = "region")

# household-level redistribution (unknown units without roster)
weighting_spec(sample_survey, base_weights = pw) |>
  step_unknown_eligibility(unknown = unknown_elig, by = "region",
                           cluster = "household_id")
```

summary.prepped_weighting_spec *Detailed per-step diagnostics*

Description

Prints the full audit of an estimated recipe: the stage-by-stage evolution of the weights, then one block per step with that step's own diagnostics table and the design effect before and after it, and finally the R-indicator when the recipe adjusted for nonresponse. Where `print()` answers "what is in this recipe?", `summary()` answers "what did each step do, and what did it cost?".

Usage

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

Arguments

`object`	a prepped object (output of prep()).
`...`	ignored.

Value

(invisibly) the prepped object.

Examples

```
fitted <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  prep()
summary(fitted)
```

transition_matrix *Gross-flow transition matrix between two panel waves*

Description

Weighted cross-tabulation of a categorical state at an earlier wave (`from`) against a later wave (`to`) – e.g. the labour-force status of the same people across two periods – computed on the **longitudinal** weight. This is the gross flow (CEPAL ch. XVII): who moved between which states. `boot_transition()` adds a per-cell standard error from the ordinary bootstrap.

Usage

```
transition_matrix(
  data,
  from,
  to,
  weights = NULL,
  states = NULL,
  format = c("row", "col", "joint", "counts")
)
```

Arguments

`data`	a `data.frame` (wide longitudinal file) or a prepped `weighting_spec`.
`from, to`	names of the earlier- and later-wave state columns.
`weights`	a weight column name or a numeric vector; defaults to 1 (or the prepped weight when `data` is a prepped spec).
`states`	optional character vector fixing the state levels and their order.
`format`	`"row"` (default, P(to|from)), `"col"` (P(from|to)), `"joint"` (P(from,to)) or `"counts"` (weighted counts).

Value

a `weightflow_transition` object holding the matrix.

Why there is a single weight, and not two

Feinberg and Stasny (1983) describe a gross-change table built from the **two cross-sectional weights**: when $w_{k,t-1} \neq w_{k,t}$, the smaller weight goes to the (i, j) cell and the difference goes to an "out of population" cell – (Outside, j) or (i, Outside) depending on the sign – on the assumption that the weights differ only because of natural entries to and exits from the target population. ECLAC cites it (ch. XVI, sec. B) as the state of things *before* a longitudinal weight is built.

This function takes **one** weight because the package builds that longitudinal weight instead, by the sequence the same chapter prescribes: panel base weight, adjustment for nonresponse in the first period, an explicit definition of the longitudinal population, the attrition adjustment and the final calibration. Once that weight exists the discrepancy Feinberg-Stasny reconstructs no longer arises – who left the population was decided explicitly at `step_drop_ineligible()` rather than inferred from a difference between two weights. The two-weight construction is the alternative to the longitudinal weight, not a complement to it.

See Also

`boot_transition()`

two_phase_variance *Decompose a two-phase variance into V = V1 + V2*

Description

For a recipe containing step_subsample(), two_phase_variance() splits the recipe-aware bootstrap variance of an estimate into its first-phase and second-phase components, $V = V_1 + V_2$. The per-unit coupling factor has variance $d = (1 - f_1)\pi_2 + (1 - \pi_2)$, the sum of the phase-1 term $(1 - f_1)\pi_2$ and the phase-2 conditional term $1 - \pi_2$; running the same bootstrap with each term in turn yields the two components.

Usage

```
two_phase_variance(
  object,
  variable,
  estimator = c("mean", "total"),
  replicates = 500L,
  seed = NULL,
  fpc = NULL
)
```

Arguments

object	a weighting_spec (or prepped) whose recipe contains step_subsample().
variable	name of the study variable (a single column).
estimator	"mean" (default) or "total".
replicates, seed, fpc	passed through to bootstrap_weights().

Details

The share prop_phase2 = V2 / V is an operational read: a large share means the second-phase subsampling drives the uncertainty, so subsampling more would pay off; a small share means a denser (more expensive) subsample would buy little, and the first phase is the binding constraint.

Value

An object of class weightflow_tp_variance: V1, V2, V, the matching standard errors se1, se2, se, and prop_phase2 = V2 / V.

See Also

bootstrap_weights(), step_subsample().

Examples

```
df <- transform(sample_survey,
                in2 = as.integer(runif(nrow(sample_survey)) < 0.3), p2 = 0.3)
spec <- weighting_spec(df, base_weights = pw) |>
  step_subsample(selected = in2, prob = p2, psu = "household_id")
two_phase_variance(spec, "income", replicates = 100)
```

---

wave_bootstrap *Coordinated bootstrap across panel waves*

Description

Builds recipe-aware bootstrap replicate weights for two or more waves that are **coordinated** by PSU: a PSU present in several waves gets the same resampling in all of them, so the sampling covariance between waves – which the overlap of a rotating panel induces – is captured. Feed the result to `change_estimate()` for the honest variance of a net change. Ignoring the coordination (a separate per-wave bootstrap) over-estimates the variance of change whenever the waves overlap and are correlated.

Usage

```
wave_bootstrap(
  specs,
  replicates = 500L,
  strata = NULL,
  psu = NULL,
  m = NULL,
  seed = NULL,
  refit_steps = "all",
  resample = c("multinom", "binom"),
  lonely_psu = c("certainty", "collapse"),
  progress = TRUE
)
```

Arguments

`specs`	a **named** list of `weighting_spec` objects, one per wave; the names are the wave labels used by `change_estimate()`.
`replicates`	number of bootstrap replicates.
`strata, psu`	column names of the design strata and PSU, present in every wave's data. The PSU id must be **consistent across waves** (the same value = the same PSU): that consistency is what makes the coordination possible. `strata = NULL` treats the whole sample as one stratum; `psu = NULL` treats each row as its own PSU.
`m`	optional PSU resample size per stratum (default `n_h - 1`, the Rao-Wu choice).
`seed`	optional integer seed for the permanent uniforms (reproducibility).

`refit_steps` which recipe steps to re-run per replicate. `"all"` (default, the honest recipe-aware bootstrap) re-preps the whole recipe, propagating the variance of every step. `"calibration"` freezes everything up to the first calibration step and re-runs only calibration per replicate – the Statistics Canada LFS convention (the Rao-Wu factor is applied to the frozen pre-calibration subweight). A character vector of step classes (e.g. `c("step_nonresponse", "step_calibrate")`) splits at the first one matched: the prefix is frozen, the suffix re-prepped. The point estimate always uses the full recipe; only the replicate variance is affected.

`resample` the Rao-Wu resample scheme. `"multinom"` (default) draws an exact multinomial per stratum (counts sum to `m_h`), the centred Rao-Wu draw, which gives an unbiased replicate variance for composite/calibration estimators and for totals; `"binom"` (legacy) draws an independent binomial per PSU (counts need not sum to `m_h`), which loses the multinomial's negative between-PSU covariance and estimates an uncentred variance – only mildly high for a ratio or mean (~+8-10%) but a large overestimate for a **total** (up to an order of magnitude), so use `"multinom"` when estimating totals. Both share the per-PSU uniforms; the multinomial coordinates across rotating waves exactly when the PSU sets match (identical or disjoint waves) and approximately otherwise.

`lonely_psu` how to treat a stratum with a single PSU in any wave. Such a stratum contributes no variance under either engine (the lone PSU cannot be resampled away or deleted), so the change SE is understated – with every stratum lonely it is exactly 0. `"certainty"` (default) leaves them alone and warns; `"collapse"` merges them into a pseudo-stratum, giving a conservative variance. The collapse map is built over all the waves and applied identically to each, so the replicate pairing that carries the overlap covariance is preserved.

`progress` show a progress message per wave.

Details

Self-contained: it does not modify `bootstrap_weights()`; each wave's recipe is re-run through `prep()` on perturbed base weights, exactly as the single-sample bootstrap does, but with a PSU-coordinated Rao-Wu draw shared across waves.

Composite (CRE) recursion is handled automatically: when a wave's recipe holds a `step_cre()` whose `previous` is the preceding wave, that wave's estimated control totals `Zhat*` are re-estimated per replicate from the previous wave's replicate-b weights (not the frozen point weights), so the reported variance includes the sampling variance of `Zhat` itself. Waves are processed in order and each wave's replicate matrix is carried into the next, giving the full coordinated variance of the recursive estimator. (Assumes each `step_cre`'s `previous` is the immediately preceding wave in `specs`; otherwise that step keeps its fixed point `previous`.)

Value

an object of class `weightflow_wave_boot`: per-wave point weights, replicate matrices (aligned by replicate index across waves), the per-wave data, and the design.

See Also

`change_estimate()`, `bootstrap_weights()`, `panel_design()`

Examples

```
t1 <- subset(panel_ine, wave == 1 & disposition == "R")
t2 <- subset(panel_ine, wave == 2 & disposition == "R")
wb <- wave_bootstrap(
  list(T1 = weighting_spec(t1, base_weights = pw),
       T2 = weighting_spec(t2, base_weights = pw)),
  replicates = 200, strata = "stratum", psu = "psu", seed = 1, progress = FALSE)
change_estimate(wb, function(w, d) weighted.mean(d$unemployed, w, na.rm = TRUE))
```

---

wave_carry *Extract the carry artifact of a period*

Description

The object `wave_step()` leaves for the next run: the PSU multiplicities that coordinate the next period's bootstrap, the replicate values of the declared estimands, and – only when the recipe needs them – the replicate weights.

Usage

```
wave_carry(x)
```

Arguments

x	a `wf_wave_step` from `wave_step()`.

Value

a `wf_wave_carry`, to be saved (`saveRDS()`) and passed as `previous` next period.

See Also

`wave_step()`

---

wave_contrast *Linear combination of an estimand across a chain of periods*

Description

`wave_step()` reports the level of its own period and the net change against each carry it was given – that is, **pairwise** contrasts. Many published series are not pairwise: a rolling quarter is the average of three consecutive months, a semester-on-semester contrast is `c(-1/2, -1/2, 1/2, 1/2)`, an annual average is `rep(1/W, W)`. `wave_contrast()` estimates any such combination $\psi = a'\theta$ directly from the carries the chain already wrote to disk, with $V(\psi) = a'\Sigma a$.

Usage

```
wave_contrast(carries, estimand, contrast = NULL, level = 0.95)
```

Arguments

carries	a list of wf_wave_carry objects, one per period entering the combination. Order defines the order of contrast; they are used as given (not sorted).
estimand	name of the estimand to combine, as it appears in carry$point (a domain estimand is named e.g. "rate|sex=F").
contrast	numeric weights, one per carry. Defaults to the average rep(1/W, W).
level	confidence level for the interval.

Details

The carries make this possible without keeping the waves in memory: each one stores the R replicate values of every declared estimand, and those replicates are **paired across periods** because the coordination transferred the PSU multiplicities. Stacking them into an R x W matrix and taking its (uncentred-at-R) covariance recovers the full between-period covariance matrix, from which any linear combination follows. With contrast = c(-1, 1) the result reproduces the $change row of wave_step() exactly.

Value

a one-row data.frame with estimate, se, V, lo, hi, R_used, the periods and the contrast used, plus the covariance matrix Sigma as an attribute.

See Also

wave_step(), wave_carry(); panel_estimate() does the same on a wave_bootstrap() object, when all waves are held together.

---

wave_jackknife *Coordinated delete-one jackknife across panel waves*

Description

Deterministic counterpart of wave_bootstrap(): builds delete-one-PSU replicate weights that are **coordinated** across waves (the same PSU is removed from every wave simultaneously), so the sampling covariance the overlap induces is captured without any randomness. Feed the result to change_estimate() for an honest, reproducible variance of a net change. Meant as the cheap exact oracle to validate wave_bootstrap(); it is also a valid variance estimator in its own right.

Usage

```
wave_jackknife(
  specs,
  strata = NULL,
  psu = NULL,
  refit_steps = "all",
  lonely_psu = c("certainty", "collapse"),
  progress = TRUE
)
```

Arguments

specs	a **named** list of weighting_spec objects, one per wave; the names are the wave labels used by change_estimate().
strata, psu	column names of the design strata and PSU, present in every wave's data. The PSU id must be **consistent across waves** (the same value = the same PSU): that consistency is what makes the coordination possible. strata = NULL treats the whole sample as one stratum; psu = NULL treats each row as its own PSU.
refit_steps	which recipe steps to re-run per replicate; see wave_bootstrap().
lonely_psu	see wave_bootstrap(). "all" (default) re-preps the whole recipe; "calibration" freezes the prefix and re-runs only calibration (StatCan LFS convention).
progress	show a progress message per wave.

Details

Self-contained: each wave's recipe is re-run through prep() on base weights with the deleted PSU zeroed and its stratum mates reweighted by $n_h/(n_h-1)$; it does not modify bootstrap_weights() or touch R/variance.R.

Value

an object of class weightflow_wave_jack accepted by change_estimate(); replicate columns are aligned across waves by the union of PSU ids (a PSU absent from a wave, or alone in its stratum, yields that wave's point weights, i.e. a zero jackknife contribution).

See Also

wave_bootstrap(), change_estimate()

Examples

```
t1 <- subset(panel_ine, wave == 1 & disposition == "R")
t2 <- subset(panel_ine, wave == 2 & disposition == "R")
wj <- wave_jackknife(
  list(T1 = weighting_spec(t1, base_weights = pw),
       T2 = weighting_spec(t2, base_weights = pw)),
  strata = "stratum", psu = "psu", progress = FALSE)
change_mean(wj, "unemployed")
```

wave_step *One period of a coordinated panel bootstrap, chained from the previous ones*

Description

Runs the whole weighting recipe of a single period, coordinates its bootstrap with the periods that share sample with it, and returns the cross-sectional weights, the level and net-change estimates, and a compact **carry** artifact for the next run. It is the production counterpart of `wave_bootstrap()`: that one needs every wave in memory at once, this one needs only what the earlier runs left behind.

Usage

```
wave_step(
  spec,
  previous = NULL,
  estimands = NULL,
  by = NULL,
  replicates = 500L,
  strata = NULL,
  psu = NULL,
  period = NULL,
  rotation_map = NULL,
  m = NULL,
  seed = NULL,
  refit_steps = "all",
  resample = c("multinom", "binom"),
  carry = c("auto", "thin", "fat"),
  progress = TRUE
)
```

Arguments

spec	a `weighting_spec()` for this period, with its full recipe.
previous	a `wf_wave_carry` from a previous run, or a list of them; `NULL` for the first period of a chain.
estimands	a named list of functions `function(w, data)` each returning one number (e.g. `list(unemp = function(w, d) weighted.mean(d$unemployed, w, na.rm = TRUE))`). Their replicate values are what the carry stores, and what the next period needs to compute a net change. Required to estimate change.
by	optional character vector of domain columns: every estimand is also evaluated within each level, and the change is reported per domain.
replicates	number of bootstrap replicates. Must be constant along the chain.
strata, psu	column names identifying the stratum and the primary sampling unit. The PSU identity is nested within the stratum, so ids restarting at 1 in each stratum are handled.

period	a label for this period (defaults to "t<seq>"). Used in the carry and in the change table.
rotation_map	optional named character vector mapping a new PSU id to the id of the PSU it replaces, in the same nested stratum + psu form. Improves the pairing; without it partners are matched in sorted order.
m	PSUs drawn per stratum (default n_h - 1).
seed	integer seed. The caller's RNG state is restored on exit.
refit_steps	which steps to re-run per replicate; "all" (default) re-runs the entire cascade, which is what makes the variance recipe-aware.
resample	"multinom" (default, exact Rao-Wu) or "binom" (legacy).
carry	"auto" (default) stores the replicate weights only when the recipe needs them, i.e. when it contains a step_cre(); "thin" forces the compact form (only the replicate values of the estimands); "fat" always stores the replicate weight matrix, which lets a later run estimate an estimand that was not declared here.
progress	print progress messages.

Details

Coordination transfers the **multiplicity** of each primary sampling unit, following the Statistics Canada LFS (cat. 71-526-X, section 7.2.2, after Roberts, Kovacevic, Mantel and Phillips 2001): current-period PSUs are paired with earlier ones, common PSUs with themselves, and each inherits its partner's multiplicity. When a stratum's PSU count is unchanged this is a permutation, so the stratum total is preserved exactly and every shared PSU keeps exactly its resampling; the coordination is then exact. When the count changes, multiplicity is added or dropped at random until the stratum total closes, and the coordination is approximate – $strata reports, per stratum, the share of replicates that needed no adjustment.

previous is a **list** of carries, not a single one, because which earlier periods share sample is a property of the rotation design: a 6-consecutive design overlaps at lags 1 to 5 and nowhere else, a 4-(8)-4 design overlaps at lags 1-3 and again at 9-15, and a 1-(3)-1-(3)-1-(3)-1 design has no overlap at lag 1 at all. Each PSU inherits from the most recent carry that holds it, so gaps and returning cohorts need no window parameter. Supply every carry whose period shares sample with this one.

The cross-sectional weights are **not** touched by any of this: $weights is exactly prep(spec)$final_weight.

Value

an object of class wf_wave_step with $weights (the publishable cross-sectional weights), $level, $change, $strata (the per-stratum coordination report), and $carry – the artifact to save for the next period.

See Also

wave_carry(), wave_bootstrap(), change_estimate()

weight_factors *Per-unit adjustment factors table*

Description

Unrolls the cascade into a `data.frame`: the weight of every unit at every stage, plus the factor each step applied to it. This is the tidy form of `prep()`'s `$history`, and the starting point for any diagnostic that plot() does not already draw.

Usage

```
weight_factors(object)
```

Arguments

`object`	a prepped object (output of prep()).

Value

data.frame with one weight column per stage and one factor per step.

See Also

Other cascade audit: `as_sae_input()`, `collect_propensities()`, `collect_step_detail()`, `collect_weights()`, `domain_summary()`, `weighting_alerts()`

Examples

```
fitted <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  prep()
head(weight_factors(fitted))
```

weightflow-alerts *Quality alerts raised while preparing a recipe*

Description

`prep()` computes non-fatal *quality alerts* at every stage of the cascade, not only at the end. Each alert names a specific risk, its trigger and a remedy. Alerts are always stored on the prepped object (read them with `weighting_alerts()` / `has_alerts()`) and shown in the HTML report; with `prep(warn = TRUE)` they are also raised as R warnings. Every alert is tagged with the step that produced it, e.g. `"[step_calibrate] ..."`.

Details

This page catalogues the alerts by theme. Thresholds are the `prep()` defaults unless noted; `min_cell_n` and `max_factor` are arguments of `prep()`.

Weight distribution

- *Negative calibration weights*: an unbounded linear/GREG calibration produced weight(s) below zero. They stay active (the totals are honest), but a negative survey weight is rarely wanted and makes the estimator erratic. Use a bounded distance (`calfun = "logit"` with `bounds`) or `step_trim_calibrated()`.
- *Sub-one weights*: unit(s) with a final weight below 1, so they represent less than themselves. Usually a sign of an over-aggressive adjustment.
- *Extreme g-factors*: a calibration adjustment factor outside the Deville-Sarndal range `[0.1, 10]`. Consider bounds, a ridge `penalty`, or coarser auxiliaries.

Adjustment cells

- *Small cells*: an adjustment cell (weighting class or post-stratum) with fewer than `min_cell_n` (default 30, Kalton and Flores-Cervantes 2003) cases. Collapse cells or switch to raking.
- *Empty cells*: a cell with no unit to adjust to (no respondents, or all of unknown eligibility); the affected units are set to weight 0.
- *Large adjustment factors*: a per-cell factor above `max_factor` (default 2.5), which inflates the variance.

Nonresponse and response propensities

- *Tiny propensities*: a minimum fitted response propensity below 0.01 blows up the `1/p` weights; check the model or trim.
- *Collapsed propensity classes*: the fitted propensities were nearly constant, so the requested `num_classes` could not be formed and the class correction did nothing.
- *Miscalibrated propensities*: a calibration slope far from 1; honest probabilities matter for `1/p`. Use `num_classes` (rank-robust) or revise the model.
- *Non-positive nonresponse g-weights*: an implied response probability of zero or below; use a bounded distance or other auxiliaries.

Calibration

- *Ill-conditioned system*: near-collinear auxiliaries (large condition number) make the weights unstable; drop a redundant auxiliary or set a ridge `penalty`.
- *Did not converge*: raking, linear or bounded calibration stopped without meeting every target; increase `maxit` or check the margins are mutually consistent.
- *Reconciled control totals*: the tidy `totals` did not all sum to the same population size and were rescaled to a common N (informative, not an error).

Machine-learning adjustments

- *Learner without cross-fitting*: a flexible learner (tree / forest / boost) run without `crossfit` can understate the variance through same-sample prediction. Add `crossfit` to estimate each unit out of sample.

Two-phase (double) sampling

- *Few phase-2 sampling units*: fewer than 30 units were subsampled at the second phase. The recipe-aware bootstrap resamples the phase-2 variance component (V2) at this level, so few units leave V2 with few degrees of freedom and an unstable phase-2 standard error. Inspect the split with `two_phase_variance()`.
- *Tiny phase-2 probability*: a minimum phase-2 selection probability below 0.02 expands the subsampled weights sharply (up to `1/pi2`), inflating V2. Check the phase-2 design or trim the expanded weights.

Finalising

- *Rounded to zero*: rounding pushed a small or negative weight to exactly 0, so the unit left the active set and `collect_weights()`; round to more decimals or resolve those weights first.

See Also

`prep()`, `weighting_alerts()`, `has_alerts()`, `step_assert()` for a hard quality gate, and `vignette("inspecting-audit` for the full programmatic quality-control workflow.

`weightflow-concepts` *Conventions shared by every weightflow step*

Description

Five things behave the same way in all twelve `step_*()` functions and are easier to learn once than twelve times: what the *active set* is and how a unit leaves it, why the order of the cascade is not arbitrary, the three different scales on which weight bounds are expressed, which arguments take a bare column name and which take a string, and how to read the diagnostics that `prep()` stores.

Details

The active set. A unit is *active* when its weight is finite and non-zero (`is.finite(w) & w != 0`). A weight of exactly `0` is the "dropped" marker: `step_drop_ineligible()` sets out-of-scope units to zero, an empty adjustment cell collapses to zero, and a unit that leaves the active set takes no part in any later step and is not returned by `collect_weights()`. A *negative* weight, by contrast, is a valid (if unusual) output of unbounded linear/GREG calibration and stays active: it is counted by `collect_weights()`, the stage funnel and `design_effect()`, so the reported totals match the weights actually returned. (One caveat: the Kish design effect assumes non-negative weights, so with negatives present its value is inflated – see `design_effect()`.)

Why the order is not arbitrary. Each step multiplies the *current* weight, i.e. the weight leaving the previous step, so the cascade is read top to bottom. The methodological order of a household survey

is: resolve unknown eligibility (step_unknown_eligibility()) while the ineligible units are still present, then drop the ineligible (step_drop_ineligible()), undo any within-cluster subsampling (step_select_within()), adjust for nonresponse (step_nonresponse()), calibrate to population totals (step_calibrate()), and finally trim, round or rescale for delivery. Putting calibration before a trim, or a trim before the nonresponse adjustment, changes the estimator, which is why the steps are explicit rather than inferred.

Three scales for weight bounds. Bounds are expressed on three different scales, and mixing them up is the most common mistake for users coming from survey:

- step_trim() max_ratio / min_ratio are a **ratio** to a reference (each unit's base weight, the group median, or an absolute value): max_ratio = 3 caps a weight at three times its reference.
- step_trim_weights() and step_trim_calibrated() lower / upper are **absolute weights**: upper = 400 caps the weight itself at 400.
- step_calibrate() bounds = c(L, U) bound the calibration **g-factor** w_new / d (the multiplicative adjustment), not the weight.

Bare column name vs string. Arguments that name a single variable to be *evaluated on the data* take an unquoted (bare) column name or condition: base_weights in weighting_spec(), and respondent, unknown, ineligible, prob, n_eligible, n_selected in the steps. Arguments that name *grouping or design structure* take character strings: by, cluster, and strata / psu in the variance functions. So it is step_nonresponse(respondent = responded, by = "region") – responded bare, "region" quoted.

The same concept, different argument names. A few concepts are named differently across functions for historical reasons. The correspondence, so you do not have to guess:

- *Weight bounds*: step_trim() uses max_ratio / min_ratio (a ratio), while step_trim_weights() and step_trim_calibrated() use upper / lower (absolute weights). See "Three scales for weight bounds" above.
- *Primary sampling unit*: as_svydesign() names it ids, while bootstrap_weights() and jackknife_weights() name it psu. Both mean the cluster identifier.
- *Calibration form vs distance*: in step_calibrate(), method selects the family ("raking", "poststratify", "linear"); calfun selects the distance function ("linear", "logit", "raking") and applies only when method = "linear". So method = "linear" and calfun = "linear" are not the same knob.

Reading the diagnostics. prep() returns an object that carries the weight at every stage ($history, a named list of vectors), one entry per step ($steps, each with its own $diagnostics table and $alerts), and the recipe-level $alerts. Rather than read those directly, use summary() for the stage-by-stage audit, weighting_alerts() / has_alerts() for the quality incidents, plot() for the visual cascade, weight_factors() for the per-unit factor table, design_effect() for the Kish design effect, and report_weighting() for the full self-contained HTML report.

`weighting_alerts` *Quality alerts recorded while preparing a recipe*

Description

`weighting_alerts()` returns the character vector of quality incidents recorded by `prep()`, each tagged with the step that produced it. `has_alerts()` is a convenience predicate. Every incident is captured here, including warnings a step raises internally (for example a calibration that could not meet its constraints), so this is the reliable channel for programmatic quality control even when warnings were suppressed.

Usage

```
weighting_alerts(object)

has_alerts(object)
```

Arguments

`object`	a prepped_weighting_spec, as returned by `prep()`.

Value

`weighting_alerts()`: a character vector (empty if the recipe ran clean). `has_alerts()`: a single logical.

See Also

Other cascade audit: `as_sae_input()`, `collect_propensities()`, `collect_step_detail()`, `collect_weights()`, `domain_summary()`, `weight_factors()`

Examples

```
fit <- weighting_spec(sample_survey, base_weights = pw) |>
  step_trim(max_ratio = 3) |>
  prep()
weighting_alerts(fit)
has_alerts(fit)
```

`weighting_spec` *Start a weighting specification*

Description

Opens a weighting recipe on a sample and its design base weights. The object it returns is inert: it holds the data, the name of the base-weight column and an empty list of steps, and computes nothing. Every `step_*()` function takes such an object and returns it with one more step appended; `prep()` estimates the result.

Usage

```
weighting_spec(data, base_weights = NULL, nonprob = FALSE)
```

Arguments

`data`	data.frame with the sample units (one row per case).
`base_weights`	unquoted name of the design base-weight column. For a non-probability sample with no design weights, leave it `NULL` and set `nonprob = TRUE`: every unit then starts with a base weight of 1.
`nonprob`	logical. Declare the sample as non-probability (an opt-in panel, a volunteer or river sample). Required when `base_weights = NULL`. The flag is recorded so the report states the sample is non-probability and adds the methodological caveat; a non-probability sample is usually adjusted with `step_pseudoweight()` (inverse participation propensity against a reference) and/or `step_calibrate()` / `step_model_calibration()` to a `reference_sample()`. A non-probability panel that already carries recruitment/base weights can pass them as `base_weights` together with `nonprob = TRUE`.

Value

an object of class "weighting_spec".

Examples

```
rec <- weighting_spec(sample_survey, base_weights = pw)
rec
# a non-probability sample: no design weights, base weight 1
np <- weighting_spec(sample_survey, base_weights = NULL, nonprob = TRUE)
```

`write_recipe` *Write a weighting recipe to a YAML file*

Description

Serializes the recipe (the weighting *method*, not the data) to a human-readable YAML file: the base-weight column, the non-probability flag, and every step's id, type and parameters. The file is a versionable metadata artifact you can review in a pull request, archive next to the quality report, or reference from a metadata system. Read it back with `read_recipe()`.

Usage

```
write_recipe(object, file, timestamp = TRUE)
```

Arguments

`object`	a `weighting_spec` (or a prepped one; only the recipe is written, never the weights or the data).
`file`	path to the `.yml`/`.yaml` file to write.
`timestamp`	whether to record the write time (UTC) in the file. Default `TRUE`; set `FALSE` for byte-identical output across writes of the same recipe (cleaner version-control diffs).

Details

Formulas and captured column expressions are stored as text. A `reference_sample()` is stored as a descriptor only (its microdata is not metadata), so a step that calibrates or pseudo-weights against a reference must be given that reference again when the recipe is reconstructed. Small control-totals tables (for example a tidy poststratification total) are serialized in full; a data frame larger than 10,000 rows is treated as microdata and rejected (route it through `reference_sample()`).

A recipe is portable only if its captured expressions reference columns of the data (for example `respondent = responded`). An expression that referenced objects from your R session (say `respondent = id %in% ids_resp`) is stored as text but cannot be reconstructed elsewhere, and will error at `prep()`.

Value

the `file` path, invisibly.

See Also

`read_recipe()`

Other recipe serialization: `read_recipe()`

Examples

```
spec <- weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_calibrate(method = "raking", margins = list(region = c(table(population$region))))
f <- tempfile(fileext = ".yml")
write_recipe(spec, f)
```

`y_model` *Specify a working model for a study variable y*

Description

Declares the working model that `step_model_calibration()` fits for one study variable: a formula, a learner (a linear/`glm` model or a machine-learning method such as a regression tree, a random forest or gradient boosting) and, for `glm`, a family. It builds no model and touches no data – it records the specification that the calibration step will fit on the sample and predict over the population.

Usage

```
y_model(formula, engine = c("glm", "tree", "forest", "boost"), family = NULL)
```

Arguments

`formula`	full formula, e.g. income ~ sex + age_g.
`engine`	"glm", "tree" (rpart), "forest" (ranger) or "boost" (xgboost). The flexible learners run with fixed default settings (hyperparameters are not currently exposed): "tree"/"forest" use the 'rpart'/'ranger' defaults, and "boost" uses xgboost with nrounds = 150, max_depth = 4 and eta = 0.1.
`family`	for engine = "glm": "gaussian", "binomial" or "poisson". For tree/forest, regression vs classification is inferred from y.

Value

a model specification list.

Examples

```
y_model(income ~ age + sex, engine = "glm")
```

Index

∗ **cascade audit**
  as_sae_input, 4
  collect_propensities, 13
  collect_step_detail, 16
  collect_weights, 17
  domain_summary, 20
  weight_factors, 97
  weighting_alerts, 101
∗ **confidentiality tools**
  disclosure_risk, 19
∗ **datasets**
  panel_datasets, 27
  population, 34
  sample_one, 43
  sample_survey, 44
∗ **diagnostics**
  nr_sensitivity, 26
∗ **non-probability tools**
  data_defect, 18
∗ **recipe serialization**
  read_recipe, 37
  write_recipe, 103
∗ **variance estimation**
  as_svydesign, 5
  bootstrap_estimate, 7
  bootstrap_weights, 9
  collect_replicate_weights, 14
  jackknife_estimate, 21
  jackknife_weights, 23
∗ **weighting steps**
  step_assert, 44
  step_calibrate, 49
  step_cre, 53
  step_drop_ineligible, 60
  step_model_calibration, 61
  step_nonresponse, 65
  step_nr_sensitivity, 68
  step_pseudoweight, 71
  step_rescale, 74
  step_round, 75
  step_select_within, 77
  step_subsample, 78
  step_trim, 80
  step_trim_calibrated, 81
  step_trim_weights, 83
  step_unknown_eligibility, 85

as_sae_input, 4
as_sae_input(), *14*, *16, 17*, *21*, *97*, *101*
as_svrepdesign *(*as_svydesign*)*, 5
as_svydesign, 5
as_svydesign(), *9, 10*, *15*, *23, 24*, *100*

boot_flows, 6
boot_mean *(*bootstrap_estimate*)*, 7
boot_total *(*bootstrap_estimate*)*, 7
boot_transition, 7
boot_transition(), *6*, *40, 41*, *87, 88*
bootstrap_estimate, 7
bootstrap_estimate(), *5, 6*, *10*, *15*, *23, 24*
bootstrap_weights, 9
bootstrap_weights(), *4–7*, *9*, *15*, *23, 24*, *39*, *41*, *51*, *79*, *89*, *91*, *94*, *100*

change_estimate, 11
change_estimate(), *26*, *30, 31*, *40, 41*, *58, 59*, *90, 91*, *93, 94*, *96*
change_mean *(*change_estimate*)*, 11
change_mean(), *12*, *40*
change_total *(*change_estimate*)*, 11
change_total(), *12*
collect_estimates, 13
collect_estimates(), *41*, *58, 59*
collect_propensities, 13
collect_propensities(), *5*, *16, 17*, *21*, *97*, *101*
collect_replicate_weights, 14
collect_replicate_weights(), *6*, *9, 10*, *20*, *23, 24*

collect_step_detail, 16
collect_step_detail(), *5*, *14*, *17*, *21*, *70*, *97*, *101*
collect_weights, 17
collect_weights(), *5*, *13, 14*, *16*, *19*, *21*, *60*, *97*, *99*, *101*

data_defect, 18
design_effect, 19
design_effect(), *5*, *21*, *99, 100*
disclosure_risk, 19
domain_summary, 20
domain_summary(), *5*, *14*, *16, 17*, *97*, *101*

has_alerts *(*weighting_alerts*)*, 101
has_alerts(), *36*, *97*, *99, 100*

jack_mean *(*jackknife_estimate*)*, 21
jack_total *(*jackknife_estimate*)*, 21
jackknife_estimate, 21
jackknife_estimate(), *6*, *9, 10*, *15*, *24*
jackknife_weights, 23
jackknife_weights(), *4–6*, *9, 10*, *15*, *23*, *39*, *100*

level_estimate, 25
level_estimate(), *12*, *58, 59*
level_mean *(*level_estimate*)*, 25
level_mean(), *26*
level_total *(*level_estimate*)*, 25
level_total(), *26*

nr_sensitivity, 26
nr_sensitivity(), *69, 70*

panel_cl *(*panel_datasets*)*, 27
panel_datasets, 27
panel_design, 28
panel_design(), *33*, *40, 41*, *57, 58*, *71*, *91*
panel_estimate, 30
panel_estimate(), *58, 59*, *93*
panel_ine *(*panel_datasets*)*, 27
panel_mean *(*panel_estimate*)*, 30
panel_merge, 32
panel_merge(), *30*
panel_pr, 33
panel_pr(), *71*
panel_puro *(*panel_datasets*)*, 27
panel_total *(*panel_estimate*)*, 30
panel_us *(*panel_datasets*)*, 27

plot(), *97*, *100*
plot.prepped_weighting_spec, 34
population, 34, *44*
prep, 35
prep(), *14*, *16*, *18*, *20, 21*, *34*, *37, 38*, *55*, *76*, *91*, *94*, *97–102*
print.weightflow_boot, 36
print.weightflow_jack, 37

read_recipe, 37
read_recipe(), *103*
reference_sample, 38
reference_sample(), *28*, *30*, *51*, *62*, *72, 73*
report_panel, 40
report_panel(), *13*
report_weighting, 41
report_weighting(), *18*, *40, 41*, *100*

sample_one, 43
sample_survey, 44
set.seed(), *76*
step_assert, 44
step_assert(), *51*, *56, 57*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85, 86*, *99*
step_attrition, 45
step_calibrate, 49
step_calibrate(), *38*, *45*, *54–56*, *61–64*, *68*, *70, 71*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85, 86*, *100*
step_cre, 53
step_cre(), *27*, *45*, *51*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85, 86*, *91*, *96*
step_cross_sectional, 56
step_domain, 58
step_domain(), *13*, *41*
step_drop_ineligible, 60
step_drop_ineligible(), *45*, *48*, *51*, *56, 57*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85, 86*, *88*, *99, 100*
step_estimate *(*step_domain*)*, 58
step_estimate(), *13*, *41*
step_filter *(*step_domain*)*, 58
step_filter(), *41*
step_longitudinal *(*step_cross_sectional*)*, 56
step_longitudinal(), *29*
step_model_calibration, 61
step_model_calibration(), *38–40*, *45*, *51*, *56*, *61*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*,

83, *85, 86*, *104*
step_nonresponse, 65
step_nonresponse(), *14*, *45*, *48*, *51*, *56*, *61*, *64*, *69–71*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85, 86*, *100*
step_nr_sensitivity, 68
step_nr_sensitivity(), *26*, *45*, *51*, *56*, *61*, *64*, *68*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85, 86*
step_panel_overlap, 70
step_panel_overlap(), *33*, *48*, *57, 58*
step_pseudoweight, 71
step_pseudoweight(), *18*, *39*, *45*, *51*, *56*, *61*, *64*, *68*, *70*, *74*, *76*, *78, 79*, *81*, *83*, *85, 86*, *102*
step_rescale, 74
step_rescale(), *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73*, *76*, *78, 79*, *81*, *83*, *85, 86*
step_round, 75
step_round(), *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73, 74*, *78, 79*, *81*, *83*, *85, 86*
step_select_within, 77
step_select_within(), *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *79*, *81*, *83*, *85, 86*, *100*
step_subsample, 78
step_subsample(), *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *78*, *81*, *83*, *85, 86*, *89*
step_transition *(*step_domain*)*, 58
step_trim, 80
step_trim(), *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *83*, *85, 86*, *100*
step_trim_calibrated, 81
step_trim_calibrated(), *20*, *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*, *85, 86*, *98*, *100*
step_trim_weights, 83
step_trim_weights(), *20*, *45*, *51*, *56*, *61*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*, *83*, *86*, *100*
step_unknown_eligibility, 85
step_unknown_eligibility(), *45*, *51*, *56*, *60, 61*, *64*, *68*, *70*, *73, 74*, *76*, *78, 79*, *81*, *83*, *85*, *100*
summary(), *34*, *100*
summary.prepped_weighting_spec, 87
summary.prepped_weighting_spec(), *21*

transition_matrix, 87
transition_matrix(), *6, 7*, *40*
two_phase_variance, 89
two_phase_variance(), *99*

wave_bootstrap, 90
wave_bootstrap(), *11, 12*, *25*, *30, 31*, *41*, *58*, *93–96*
wave_carry, 92
wave_carry(), *93*, *96*
wave_contrast, 92
wave_jackknife, 93
wave_jackknife(), *25*, *30, 31*, *41*, *58*
wave_step, 95
wave_step(), *92, 93*
weight_factors, 97
weight_factors(), *5*, *14*, *16, 17*, *21*, *100, 101*
weightflow-alerts, *36*, 97
weightflow-concepts, 99
weighting_alerts, 101
weighting_alerts(), *5*, *14*, *16, 17*, *21*, *36*, *97*, *99, 100*
weighting_spec, 102
weighting_spec(), *35*, *95*, *100*
write_recipe, 103
write_recipe(), *37, 38*

y_model, 104