Package 'weightflow'

August 19, 2026

Title Declarative Recipes for Staged Survey Weighting with Recipe-Aware Replicate Variances

Version 1.1.0

Description Builds survey analysis weights by declaring the whole weighting process as an ordered recipe of explicit adjustments and estimating it in a single call. The recipe follows the stages of a real household survey: redistribution of unresolved eligibility, within-household selection, nonresponse adjustment by weighting classes, by response-propensity models (logistic regression or machine-learning learners such as trees, random forests and gradient boosting, with optional cross-fitting) or by two-phase calibration for nonresponse, calibration to known population totals following Deville and Sarndal (1992) <doi:10.2307/2290268>, optional model-assisted (model) calibration to auxiliary predictions following Wu and Sitter (2001) <doi:10.1198/016214501750333054>, and range-restricted trimming that bounds the weights while preserving the calibration totals. Variances come from a recipe-aware bootstrap and jackknife that resample or delete primary sampling units and re-apply the entire cascade on each replicate, following the rescaling bootstrap of Rao and Wu (1988) <doi:10.1080/01621459.1988.10478591>, so the replicate weights carry the variability of every adjustment instead of treating the adjustments as known. A self-contained HTML report documents each step with its diagnostics, and the weights bridge to the 'survey' and 'srvyr' packages for design-based inference.

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

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

NeedsCompilation no

Author Juan Pablo Ferreira [aut, cre] (ORCID: <https://orcid.org/0000-0002-1884-8187>), Andrés Gutiérrez [aut] (ORCID: <https://orcid.org/0009-0007-2918-1932>)

Maintainer Juan Pablo Ferreira <juanpablo.ferreira@fcea.edu.uy>

Repository CRAN

Date/Publication 2026-08-19 11:20:02 UTC

Contents

`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.

`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)

boot_total(boot, variable)

boot_mean(boot, variable)
```

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 (normal) interval.
`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`.

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,
  lonely_psu = c("certainty", "collapse"),
  seed = NULL,
  cores = 1L,
  progress = TRUE
)
```

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).
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).

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.

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")
```

`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()`

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
)
```

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).

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".

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 integer, which step to inspect (1 for the first piped step). 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()

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.

Examples

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

---

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)
```

---

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)
```

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.

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`. 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()`

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)

jack_total(jack, variable)

jack_mean(jack, variable)
```

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 (normal) interval.
`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.

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,
  lonely_psu = c("certainty", "collapse"),
  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.
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.
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).

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.

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")
```

---

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.

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.

`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")
)
```

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".

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.

Examples

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

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.

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.

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",
                 margins = list(sex    = c(table(population$sex)),
                                region = c(table(population$region)))) |>
  prep()

# 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_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)
```

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.

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.

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-assisted 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",
  cluster = NULL,
  equal_within_cluster = FALSE,
  crossfit = NULL,
  crossfit_seed = 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). 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.
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.
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.

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.

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
)
```

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.

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.

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_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)
```

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.

Value

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

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") or with the largest-remainder method ("preserve_total"), which keeps the weighted total exactly. 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"))
```

Arguments

spec	a weighting_spec.
digits	integer. Decimals to keep (0 = integers).
method	"nearest" (simple rounding) or "preserve_total" (keeps the sum of weights). Note: "preserve_total" can break equality of weights within a cluster; if you need integer and equal weights per household, use "nearest".

Value

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

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)
```

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`.

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.

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_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
)
```

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.
`min_ratio`	number or NULL. Lower floor (same units as 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.

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.

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
)
```

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()`.

`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.

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()`: it acts on the positive incoming weights and leaves dropped units (weight 0) alone.

Value

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

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
)
```

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.
`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.

Value

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

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)
```

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.

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.

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)
```

`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.

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, `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_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)
```

Arguments

`data`	data.frame with the sample units (one row per case).
`base_weights`	unquoted name of the design base-weight column.

Value

an object of class "weighting_spec".

Examples

```
rec <- weighting_spec(sample_survey, base_weights = pw)
rec
```

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.

Examples

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

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