

Package ‘weightflow’

August 30, 2026

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

Version 1.2.0

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. Steps include within-cluster selection, second-phase subsampling for two-phase sampling, nonresponse adjustment by weighting classes or response-propensity models (including machine-learning learners with optional cross-fitting), calibration to known totals following Deville and Sarndal (1992) <doi:10.2307/2290268> with optional model-assisted calibration following Wu and Sitter (2001) <doi:10.1198/016214501750333054>, adjustment of non-probability samples by pseudo-weighting, mass imputation and doubly robust estimators, and range-restricted trimming. 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 Rao and Wu (1988) <doi:10.1080/01621459.1988.10478591>, and are separated into first- and second-phase components ($V = V1 + V2$) for two-phase designs. A self-contained HTML report documents each step, and the weights bridge to the 'survey' and 'srvyr' packages.

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

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, cph] (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 2026-08-30 03:30:02 UTC

Contents

as_sae_input	3
as_svydesign	4
bootstrap_estimate	5
bootstrap_weights	7
collect_propensities	9
collect_replicate_weights	10
collect_step_detail	11
collect_weights	12
data_defect	13
design_effect	14
disclosure_risk	15
domain_summary	16
jackknife_estimate	17
jackknife_weights	18
nr_sensitivity	20
plot.prepped_weighting_spec	20
population	21
prep	22
print.weightflow_boot	23
print.weightflow_jack	23
read_recipe	24
reference_sample	25
report_weighting	26
sample_one	27
sample_survey	28
step_assert	29
step_calibrate	30
step_drop_ineligible	34
step_model_calibration	35
step_nonresponse	38
step_nr_sensitivity	42
step_pseudoweight	43

step_rescale	45
step_round	46
step_select_within	47
step_subsample	49
step_trim	50
step_trim_calibrated	52
step_trim_weights	54
step_unknown_eligibility	55
summary.prepped_weighting_spec	57
two_phase_variance	57
weightflow-alerts	59
weightflow-concepts	60
weighting_alerts	62
weighting_spec	63
weight_factors	64
write_recipe	64
y_model	65

Index 67

as_sae_input	<i>Direct estimates and design SEs per domain, ready for small-area estimation</i>
--------------	--

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 <code>c(0.165, 0.33)</code> , 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\(\)](#)

bootstrap_estimate	<i>Bootstrap estimate, standard error and confidence interval</i>
--------------------	---

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)

boot_mean(boot, variable)
```

Arguments

<code>boot</code>	a <code>weightflow_boot</code> object.
<code>statistic</code>	a function <code>function(w, data)</code> returning a numeric scalar (or vector) given a weight vector and the data.
<code>level</code>	confidence level for the interval.
<code>ci_type</code>	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).
<code>df</code>	degrees of freedom for the t interval; NULL (default) uses the design df stored on the object (total PSUs minus strata).
<code>variable</code>	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),

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

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	<i>Recipe-aware bootstrap replicate weights</i>
-------------------	---

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 <code>f_h</code> (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); <code>f_h = 0</code> 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 <code>cores > 1</code> the replicate re-preps run in parallel via <code>parallel::mclapply</code> (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 <code>two_phase_variance()</code> 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)
```

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

<code>object</code>	a prepped <code>weighting_spec</code> (the output of <code>prep()</code>).
<code>step</code>	optional integer, which step to read when the recipe has more than one propensity step. If <code>NULL</code> (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

<code>object</code>	a <code>weightflow_boot</code> or <code>weightflow_jack</code> object.
<code>weight_name</code>	name of the point-weight column to add.
<code>prefix</code>	prefix for the replicate-weight columns (<code>rep_1</code> , <code>rep_2</code> , ...).
<code>drop_zero</code>	keep only active units (point weight > 0).
<code>scramble</code>	disclosure control for a public-use file. When <code>TRUE</code> , the replicate columns are randomly permuted (their <code>rscales</code> move with them, so the variance is unchanged) and the design identifier columns (the <code>strata</code> and <code>psu</code> 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 <code>"scrambled" = TRUE</code> .

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	<i>Per-unit detail of one step of the cascade</i>
---------------------	---

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	<i>Data-defect diagnostics for a non-probability sample</i>
-------------	---

Description

For a non-probability sample, Meng (2018) decomposes the error of the sample mean into the data-defect correlation (ρ , 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

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

which does not depend on the outcome except through ρ . Because ρ on the target variable is not observable from the sample alone, `data_defect()` returns the effective size across a grid of plausible residual ρ (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 <code>weighting_spec(..., nonprob = TRUE)</code>).
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	<i>Flag re-identification risk from outlier weights within a publication cell</i>
-----------------	---

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

<code>object</code>	a prepped <code>weighting_spec</code> (the output of <code>prep()</code>).
<code>by</code>	the name(s) of the publication cell column(s) (e.g. "region", or <code>c("region", "sex")</code>). The risk is judged within each cell.
<code>ratio</code>	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	<i>Per-domain weight summary at every stage of the cascade</i>
----------------	--

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 <code>c("region", "area")</code> 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	<i>Jackknife estimate, standard error and confidence interval</i>
--------------------	---

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)

jack_mean(jack, variable)
```

Arguments

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

Details

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

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

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	<i>Recipe-aware delete-a-PSU jackknife replicate weights</i>
-------------------	--

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 <code>weighting_spec</code> (inert recipe) or a prepped <code>weighting_spec</code> . Pass the recipe <i>before</i> <code>prep()</code> : the jackknife preps it once per replicate.
strata	name of the stratum column, or <code>NULL</code> for a single stratum.
psu	name of the PSU column, or <code>NULL</code> 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 <code>cores > 1</code> the replicate re-preps run in parallel via <code>parallel::mclapply</code> (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.

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")
```

nr_sensitivity	<i>Read the nonresponse-sensitivity analysis from a prepped recipe</i>
----------------	--

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 <code>step_nr_sensitivity()</code> .
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()`

plot.prepped_weighting_spec	<i>Diagnostic plots for the weights</i>
-----------------------------	---

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

<code>x</code>	a prepped object (output of <code>prep()</code>).
<code>type</code>	"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.
<code>...</code>	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	<i>Synthetic target population (sampling frame)</i>
------------	---

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

<code>spec</code>	a <code>weighting_spec</code> .
<code>min_cell_n</code>	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 <code>NULL</code> to disable.
<code>max_factor</code>	numeric. Adjustment factor above which a cell is flagged as excessive. Default 2.5. Set to <code>NULL</code> to disable.
<code>warn</code>	logical. If <code>TRUE</code> , the quality alerts are also raised as R warnings during <code>prep()</code> . Default <code>FALSE</code> : alerts are always computed, stored on the object (<code>\$alerts</code>) 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	<i>Read a weighting recipe from a YAML file</i>
-------------	---

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)
```

Arguments

<code>file</code>	path to the recipe .yaml/.yml file.
<code>data</code>	optional data frame. When supplied, the recipe is rebuilt into a <code>weighting_spec</code> on this data. The columns the steps reference (weights, auxiliaries, disposition flags) must exist in data.
<code>references</code>	optional named list of <code>reference_sample()</code> objects, named by the id of the step that uses each one, to restore the steps that calibrate or pseudo-weight against a reference (whose microdata the recipe does not store).

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 = ".yaml"); write_recipe(spec, f)
read_recipe(f) # inspect the manifest
spec2 <- read_recipe(f, data = sample_survey) # rebuild an executable recipe
```

reference_sample	<i>Use a weighted survey as the calibration reference instead of a frame</i>
------------------	--

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

<code>data</code>	a <code>data.frame</code> of reference-survey microdata, with the columns used in <code>x_formula</code> and the model predictors.
<code>weights</code>	either the name (string) of a positive weight column in <code>data</code> , or a numeric vector with one weight per row.
<code>replicates</code>	optional numeric matrix (or <code>data.frame</code>) of replicate weights for the reference survey – one row per reference unit, one column per replicate – used to propagate the reference sampling variance through <code>bootstrap_weights()</code> . <code>NULL</code> (default) treats the totals as fixed. Note that only <code>bootstrap_weights()</code> pairs the reference replicates and propagates this variance; <code>jackknife_weights()</code> treats the estimated totals as fixed even when <code>replicates</code> 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 Erquiurescu 2021), so the extra variance from estimating the totals is captured. Without `replicates` the totals are treated as fixed (a reasonable approximation when the reference is much larger than the sample, and the same assumption made when calibrating to another survey's published totals).

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_weighting	<i>Self-contained HTML quality report for a weighting recipe</i>
------------------	--

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 <code>prep()</code>).
file	output path; if <code>NULL</code> , a temporary <code>.html</code> 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	<i>Assert quality conditions on the weights</i>
-------------	---

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_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_calibrate	<i>Calibration to population totals</i>
----------------	---

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 <code>collect_step_detail()</code> ; 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 $c_j \rightarrow \infty$ the constraint is met exactly, as $c_j \rightarrow 0$ 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_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_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, id = NULL)
```

Arguments

<code>spec</code>	a <code>weighting_spec</code> .
<code>ineligible</code>	a 0/1 dummy column (1 = ineligible) or any logical condition (unquoted) that is TRUE for out-of-scope units.
<code>id</code>	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in <code>collect_step_detail()</code> ; defaults to a derived " <code><class>_<k></code> ".

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_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-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,
  id = NULL
)
```

Arguments

spec	a <code>weighting_spec</code> .
x_formula	formula of the consistency auxiliaries, e.g. <code>~ sex + region</code> .

models	named list of models created with <code>y_model()</code> . 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 <code>reference_sample()</code> , 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 (<code>x_formula</code>), 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 <code>step_calibrate(method = "linear")</code> : 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 <code>x_formula</code> 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 <code>x_totals</code> data frames. Only used when <code>x_totals</code> 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 ($K \geq 2$ 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 <code>collect_step_detail()</code> ; defaults to a derived " <code><class>_<k></code> ".

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](https://doi.org/10.1198/016214501750333054).

See Also

Other weighting steps: `step_assert()`, `step_calibrate()`, `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	<i>Nonresponse adjustment</i>
------------------	-------------------------------

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 <i>mass of clusters</i> 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 $K \geq 2$), 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) non-response 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_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	<i>Sensitivity of a mean to nonignorable nonresponse or selection</i>
---------------------	---

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 ϕ 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 ϕ , 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 <code>weighting_spec</code> .
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. <code>~ region + sex + age</code> .
respondent	optional response/participation indicator (bare column or condition). Defaults to <code>!is.na(y)</code> .
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 ρ (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_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_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; Elliott and Valliant 2017), which inflates each unit to the population so the weights sum to the reference's estimated population size. 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,
  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 <code>reference_sample(replicates =)</code> 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. <code>~ sex + age + region</code> .
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.
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

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_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 <code>collect_step_detail()</code> ; 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_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	<i>Round the final weights</i>
------------	--------------------------------

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"),
  id = NULL
)
```

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".
id	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in <code>collect_step_detail()</code> ; 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_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	<i>Within-cluster selection adjustment</i>
--------------------	--

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

<code>spec</code>	a <code>weighting_spec</code> .
<code>prob</code>	unquoted column with the within-household selection probability of the selected person (need not be $1/n_{\text{eligible}}$). The weight is multiplied by $1/\text{prob}$.
<code>n_eligible</code>	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 <code>n_eligible</code> (equivalent to $\text{prob} = 1/n_{\text{eligible}}$).
<code>n_selected</code>	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 $n_{\text{eligible}} / n_{\text{selected}}$ (equivalent to $\text{prob} = n_{\text{selected}}/n_{\text{eligible}}$). Defaults to 1. Only used together with <code>n_eligible</code> .
<code>id</code>	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in <code>collect_step_detail()</code> ; defaults to a derived " <code><class>_<k></code> ".

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_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	<i>Second-phase subsampling (two-phase sampling)</i>
----------------	--

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 $1/\text{prob}$. 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 <code>collect_step_detail()</code> ; defaults to a derived " <code><class>_<k></code> ".

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_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_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().
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 <code>step_calibrate(equal_within_cluster = TRUE)</code>); 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 <code>collect_step_detail()</code> ; defaults to a derived " <code><class>_<k></code> ".

Details

The absolute-weight bound is imposed as a per-unit factor bound w_{new} / w in $[\text{lower}/w, \text{upper}/w]$ 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 active incoming weights (including any negative weights an unbounded linear calibration produced, which it can bring back into $[\text{lower}, \text{upper}]$) 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.

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](https://doi.org/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_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	<i>Automatic weight trimming to an absolute band</i>
-------------------	--

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,
  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.
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()).

<code>strict</code>	logical. If TRUE (default), iterate cap+redistribution until no weight is outside [lower, upper] (like survey's <code>strict = TRUE</code>). If FALSE, a single pass (redistribution may push some weights slightly past the cap).
<code>maxit</code>	integer. Maximum iterations when <code>strict = TRUE</code> .
<code>id</code>	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in <code>collect_step_detail()</code> ; 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_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

<code>spec</code>	a <code>weighting_spec</code> .
<code>unknown</code>	a 0/1 dummy column (1 = eligibility unknown) or any logical condition (unquoted) that is TRUE for unknown-eligibility cases. Evaluated on the data.
<code>by</code>	character. Variables defining the adjustment cells (optional).
<code>cluster</code>	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.
<code>id</code>	optional string: a stable identifier for this step, shown in the recipe print-out and usable to select it in <code>collect_step_detail()</code> ; defaults to a derived " <code><class>_<k></code> ".

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_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

<code>object</code>	a prepped object (output of <code>prep()</code>).
<code>...</code>	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)
```

two_phase_variance

Decompose a two-phase variance into $V = V_1 + V_2$

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 $\text{prop_phase2} = V2 / V$ 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 $\text{prop_phase2} = V2 / V$.

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)
```

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 $1/\pi^2$), 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	<i>Quality alerts recorded while preparing a recipe</i>
------------------	---

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	<i>Start a weighting specification</i>
----------------	--

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

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

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)
```

weight_factors	<i>Per-unit adjustment factors table</i>
----------------	--

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))
```

write_recipe	<i>Write a weighting recipe to a YAML file</i>
--------------	--

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 <code>weighting_spec</code> (or a prepped one; only the recipe is written, never the weights or the data).
file	path to the <code>.yaml/.yml</code> file to write.
timestamp	whether to record the write time (UTC) in the file. Default <code>TRUE</code> ; set <code>FALSE</code> 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 = ".yaml")
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. <code>income ~ sex + age_g</code> .
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 <code>nrounds = 150</code> , <code>max_depth = 4</code> and <code>eta = 0.1</code> .
family	for engine = "glm": "gaussian", "binomial" or "poisson". For tree/forest, regression vs classification is inferred from <code>y</code> .

Value

a model specification list.

Examples

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

Index

- * **cascade audit**
 - as_sae_input, 3
 - collect_propensities, 9
 - collect_step_detail, 11
 - collect_weights, 12
 - domain_summary, 16
 - weight_factors, 64
 - weighting_alerts, 62
- * **confidentiality tools**
 - disclosure_risk, 15
- * **datasets**
 - population, 21
 - sample_one, 27
 - sample_survey, 28
- * **diagnostics**
 - nr_sensitivity, 20
- * **non-probability tools**
 - data_defect, 13
- * **recipe serialization**
 - read_recipe, 24
 - write_recipe, 64
- * **variance estimation**
 - as_svydesign, 4
 - bootstrap_estimate, 5
 - bootstrap_weights, 7
 - collect_replicate_weights, 10
 - jackknife_estimate, 17
 - jackknife_weights, 18
- * **weighting steps**
 - step_assert, 29
 - step_calibrate, 30
 - step_drop_ineligible, 34
 - step_model_calibration, 35
 - step_nonresponse, 38
 - step_nr_sensitivity, 42
 - step_pseudoweight, 43
 - step_rescale, 45
 - step_round, 46
 - step_select_within, 47
 - step_subsample, 49
 - step_trim, 50
 - step_trim_calibrated, 52
 - step_trim_weights, 54
 - step_unknown_eligibility, 55
- as_sae_input, 3
- as_sae_input(), 10, 12, 13, 16, 62, 64
- as_svrepdesign(as_svydesign), 4
- as_svydesign, 4
- as_svydesign(), 6, 8, 11, 18, 19, 61
- boot_mean(bootstrap_estimate), 5
- boot_total(bootstrap_estimate), 5
- bootstrap_estimate, 5
- bootstrap_estimate(), 4, 5, 8, 11, 18, 19
- bootstrap_weights, 7
- bootstrap_weights(), 3–6, 11, 18, 19, 25, 32, 49, 50, 58, 61
- collect_propensities, 9
- collect_propensities(), 4, 12, 13, 16, 62, 64
- collect_replicate_weights, 10
- collect_replicate_weights(), 5, 6, 8, 15, 18, 19
- collect_step_detail, 11
- collect_step_detail(), 4, 10, 13, 16, 62, 64
- collect_weights, 12
- collect_weights(), 4, 9–12, 14, 16, 34, 60–62, 64
- data_defect, 13
- design_effect, 14
- design_effect(), 4, 16, 61, 62
- disclosure_risk, 15
- domain_summary, 16
- domain_summary(), 4, 9, 10, 12, 13, 62, 64
- has_alerts(weighting_alerts), 62
- has_alerts(), 22, 59, 60, 62

- jack_mean(jackknife_estimate), 17
- jack_total(jackknife_estimate), 17
- jackknife_estimate, 17
- jackknife_estimate(), 5, 6, 8, 11, 19
- jackknife_weights, 18
- jackknife_weights(), 3–6, 8, 11, 18, 25, 61

- nr_sensitivity, 20
- nr_sensitivity(), 42, 43

- plot(), 62, 64
- plot.prepped_weighting_spec, 20
- population, 21, 28
- prep, 22
- prep(), 9, 11, 12, 14–16, 20, 24, 59, 60, 62, 63
- print.weightflow_boot, 23
- print.weightflow_jack, 23

- read_recipe, 24
- read_recipe(), 64, 65
- reference_sample, 25
- reference_sample(), 32, 36, 44, 45
- report_weighting, 26
- report_weighting(), 14, 62

- sample_one, 27
- sample_survey, 28
- step_assert, 29
- step_assert(), 32, 35, 37, 41, 43, 45–48, 50, 51, 53, 55, 56, 60
- step_calibrate, 30
- step_calibrate(), 30, 35, 37, 41, 43–48, 50, 51, 53, 55, 56, 61
- step_drop_ineligible, 34
- step_drop_ineligible(), 30, 32, 37, 41, 43, 45–48, 50, 51, 53, 55, 56, 61
- step_model_calibration, 35
- step_model_calibration(), 25, 26, 30, 32, 35, 41, 43–48, 50, 51, 53, 55, 56, 65
- step_nonresponse, 38
- step_nonresponse(), 10, 30, 32, 35, 37, 42, 43, 45–48, 50, 51, 53, 55, 56, 61
- step_nr_sensitivity, 42
- step_nr_sensitivity(), 20, 30, 32, 35, 37, 41, 45–48, 50, 51, 53, 55, 56
- step_pseudoweight, 43
- step_pseudoweight(), 14, 30, 32, 35, 37, 41, 43, 46–48, 50, 51, 53, 55, 56, 63
- step_rescale, 45
- step_rescale(), 30, 32, 35, 37, 41, 43, 45, 47, 48, 50, 51, 53, 55, 56
- step_round, 46
- step_round(), 30, 32, 35, 37, 41, 43, 45, 46, 48, 50, 51, 53, 55, 56
- step_select_within, 47
- step_select_within(), 30, 32, 35, 37, 41, 43, 45–47, 50, 51, 53, 55, 56, 61
- step_subsample, 49
- step_subsample(), 30, 32, 35, 37, 41, 43, 45–48, 51, 53, 55–58
- step_trim, 50
- step_trim(), 30, 32, 35, 37, 41, 43, 45–48, 50, 53, 55, 56, 61
- step_trim_calibrated, 52
- step_trim_calibrated(), 15, 30, 32, 35, 37, 41, 43, 45–48, 50, 51, 55, 56, 59, 61
- step_trim_weights, 54
- step_trim_weights(), 15, 30, 32, 35, 37, 41, 43, 45–48, 50, 51, 53, 56, 61
- step_unknown_eligibility, 55
- step_unknown_eligibility(), 30, 32, 34, 35, 37, 41, 43, 45–48, 50, 51, 53, 55, 61
- summary(), 20, 62
- summary.prepped_weighting_spec, 57
- summary.prepped_weighting_spec(), 16

- two_phase_variance, 57
- two_phase_variance(), 60

- weight_factors, 64
- weight_factors(), 4, 9, 10, 12, 13, 16, 62
- weightflow-alerts, 22, 59
- weightflow-concepts, 60
- weighting_alerts, 62
- weighting_alerts(), 4, 10, 12, 13, 16, 22, 59, 60, 62, 64
- weighting_spec, 63
- weighting_spec(), 22, 61
- write_recipe, 64
- write_recipe(), 24

- y_model, 65