

Package ‘weightflow’

August 4, 2026

Title Declarative API for Staged Survey Weights

Version 1.0.0

Description Builds survey weights from design base weights by chaining the stages of a weighting workflow (unknown-eligibility redistribution, nonresponse adjustment, calibration to known population totals, and weight trimming) through a declarative, pipeable, 'tidymodels'-style API, with nonresponse handled by weighting classes, by response-propensity models fitted with logistic regression or machine-learning learners (trees, random forests and gradient boosting), or by calibration. Calibration follows Deville and Sarndal (1992) <doi:10.2307/2290268>, and a range-restricted variant trims the weights into a fixed interval while preserving the calibration totals, following the generalized exponential method of Folsom and Singh (2000). Variances are obtained with a recipe-aware bootstrap and jackknife that resample primary sampling units and re-apply the whole cascade on each replicate, following the rescaling bootstrap of Rao and Wu (1988) <doi:10.1080/01621459.1988.10478591>, so the replicate weights carry the variability of every adjustment. A self-contained HTML report documents each step with diagnostics, and the weights bridge to the 'survey' and 'srvyr' packages for design-based inference.

License MIT + file LICENSE

Encoding UTF-8

Language en-US

Depends R (>= 4.1.0)

Imports stats, utils, graphics, parallel

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

Config/roxygen2/version 8.0.0

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

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

Config/testthat/edition 3

LazyData true

VignetteBuilder knitr

NeedsCompilation no

Author Juan Pablo Ferreira [aut, cre],
 Andrés Gutiérrez [ctb] (ORCID: <<https://orcid.org/0009-0007-2918-1932>>,
 affiliation: ECLAC - Statistics Division)

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

Repository CRAN

Date/Publication 2026-08-04 17:10:02 UTC

Contents

as_svydesign	3
bootstrap_estimate	3
bootstrap_weights	4
collect_replicate_weights	5
collect_weights	6
design_effect	7
jackknife_estimate	8
jackknife_weights	9
plot.prepped_weighting_spec	10
population	11
prep	11
report_weighting	12
sample_one	14
sample_survey	15
step_assert	15
step_calibrate	16
step_drop_ineligible	19
step_model_calibration	20
step_nonresponse	22
step_rescale	26
step_round	26
step_select_within	27
step_trim	28
step_trim_calibrated	29
step_trim_weights	31
step_unknown_eligibility	32
summary.prepped_weighting_spec	33
weighting_spec	34
weight_factors	34
y_model	35

Index

36

as_svydesign	<i>Export weightflow weights to a survey design</i>
--------------	---

Description

as_svydesign() builds a linearization (ultimate-cluster) design from a prepped recipe; as_svrepdesign() builds a replicate-weights design from a bootstrap (weightflow_boot) or jackknife (weightflow_jack) object, so survey/srvyr standard errors include the recipe's adjustments. Both require the 'survey' package. With replicate weights you can then estimate any statistic for any domain (svytotal, svymean, svyratio, svyby, ...) with variances that reflect the whole recipe.

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.

Value

A survey.design / svyrep.design object.

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

Description

Applies a statistic to the point weights and to every replicate, and summarises it with the bootstrap variance $(1/B) \sum (\theta_b^* - \hat{\theta})^2$.

Usage

```
bootstrap_estimate(boot, statistic, level = 0.95)

boot_total(boot, variable)

boot_mean(boot, variable)
```

Arguments

boot	a weightflow_boot object.
statistic	a function function(w, data) returning a numeric scalar (or vector) given a weight vector and the data.
level	confidence level for the (normal) interval.
variable	name of the variable to estimate.

Value

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

bootstrap_weights	<i>Bootstrap replicate weights that re-apply the recipe</i>
-------------------	---

Description

Builds bootstrap replicate weights by resampling primary sampling units (PSUs) with replacement within strata and re-running the whole recipe on each replicate. Because every adjustment (nonresponse, calibration, ...) is recomputed per replicate, the resulting replicate weights propagate the variability introduced by each weighting stage.

Usage

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

Arguments

object	a weighting_spec (or a prepped one) holding the recipe.
replicates	number of bootstrap replicates.
strata, psu	column names of the stratum and the PSU. If psu is NULL each unit is its own PSU; if strata is NULL a single stratum is assumed.
m	PSUs drawn per stratum (default n - 1).

lonely_psu	how to treat strata with a single PSU (which a with-replacement bootstrap cannot resample): "certainty" (default) treats them as self-representing, so they contribute no bootstrap variance, and warns; "collapse" merges the single-PSU strata into a pseudo-stratum (with the smallest other stratum if there is only one), so they are resampled and do contribute a (conservative) variance. For full control, build your own collapsed stratum column and pass it as strata.
seed	optional RNG seed.
cores	number of parallel workers for the replicates (default 1 = serial). With cores > 1 the replicate re-preps run in parallel via <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).

Details

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

Value

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

Examples

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

collect_replicate_weights

Collect replicate weights into a data frame ready for srvyr

Description

Returns the data with the point weight and the bootstrap replicate weights as columns, so it can be fed directly to `srvyr::as_survey_rep()` (or `survey::svrepdesign()`). Replicate columns are full weights, so use `combined.weights = TRUE`, `scale = 1 / R`, `rscales = 1`, `mse = TRUE`.

Usage

```
collect_replicate_weights(
  boot,
  weight_name = ".weight",
  prefix = "rep_",
  drop_zero = TRUE
)
```

Arguments

<code>boot</code>	a <code>weightflow_boot</code> object.
<code>weight_name</code>	name of the point-weight column to add.
<code>prefix</code>	prefix for the replicate-weight columns (rep_1, rep_2, ...).
<code>drop_zero</code>	keep only active units (point weight > 0).

Value

A data frame: the original columns, `weight_name`, and one column per replicate. The number of replicates is stored in attribute "R".

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)

if (requireNamespace("srvyr", quietly = TRUE) &&
  requireNamespace("dplyr", quietly = TRUE)) {
  srvyr::as_survey_rep(df, weights = .weight,
    repweights = dplyr::starts_with("rep_"),
    type = "bootstrap", combined.weights = TRUE,
    scale = 1 / attr(df, "R"), rscales = 1, mse = TRUE)
}
```

`collect_weights`

Extract the data with the computed weights

Description

Extract the data with the computed weights

Usage

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

Arguments

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

drop_zero logical. If TRUE, drops rows with final weight 0 (ineligible / nonresponse). Default TRUE.

keep_intermediate logical. If TRUE, adds one column per stage.

weight_name name of the final weight column. Default ".weight".

Value

data.frame.

Examples

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

design_effect	<i>Kish design effect from unequal weighting</i>
---------------	--

Description

$deff = 1 + CV^2(w) = m * \sum(w^2) / (\sum(w))^2$, over the active weights. The effective sample size is $n_{eff} = m / deff$.

Usage

```
design_effect(w)
```

Arguments

w vector of weights (zeros are dropped).

Value

list with deff, n_eff, cv and n.

Examples

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

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 summarises it with the stratified jackknife (JKn) variance

$$\sum_h \frac{n_h - 1}{n_h} \sum_{i \in h} (\theta_{(hi)} - \bar{\theta}_h)^2,$$

where $\theta_{(hi)}$ is the estimate with PSU i of stratum h deleted and $\bar{\theta}_h$ the mean of those over the stratum. No finite population correction is applied.

Usage

```
jackknife_estimate(jack, statistic, level = 0.95)
```

```
jack_total(jack, variable)
```

```
jack_mean(jack, variable)
```

Arguments

jack	a weightflow_jack object.
statistic	a function function(w, data) returning a numeric scalar (or vector) given a weight vector and the data.
level	confidence level for the (normal) interval.
variable	name of the variable to estimate (for jack_total/jack_mean).

Value

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

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>Delete-a-PSU jackknife replicate weights that re-apply the recipe</i>
-------------------	--

Description

Builds jackknife replicate weights by deleting one primary sampling unit (PSU) at a time and re-running the whole recipe on each replicate, so the replicate weights carry the variability of every adjustment (like `bootstrap_weights()`, but with the delete-a-PSU jackknife instead of a resampling bootstrap).

Usage

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

Arguments

<code>object</code>	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.
<code>strata</code>	name of the stratum column, or <code>NULL</code> for a single stratum.
<code>psu</code>	name of the PSU column, or <code>NULL</code> to delete one unit at a time.
<code>lonely_psu</code>	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.
<code>cores</code>	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.
<code>progress</code>	print progress every 25 replicates (serial only).

Details

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

Value

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

Examples

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

`plot.prepped_weighting_spec`

Diagnostic plots for the weights

Description

Diagnostic plots for the weights

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>Example target population for weightflow</i>
------------	---

Description

A simulated population (sampling frame) of individuals nested in households and primary sampling units (PSUs) within strata (regions), with demographic auxiliaries and two outcomes. Used to illustrate calibration targets and model calibration, and to validate weighted estimates. Generated by `data-raw/weightflow_data.R`.

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	<i>Estimate the weighting cascade</i>
------	---------------------------------------

Description

Walks the steps in the order they were added, starting from the base weights. Each step multiplies the current weight by its adjustment factor.

Usage

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

Arguments

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

Value

a "prepped_weighting_spec" object.

Examples

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

report_weighting	<i>Build a nice HTML report of the weighting recipe</i>
------------------	---

Description

Writes a self-contained HTML file (no dependencies, no server) showing the pipeline, the parameters requested at each step, the per-stage summary (n, sum, CV, Kish deff, effective n) and per-step diagnostics, and opens it in the browser.

Usage

```
report_weighting(
  object,
  file = NULL,
  open = TRUE,
  plots = TRUE,
  narrative = TRUE,
  lang = c("en", "es"),
  metadata = NULL,
  replicates = NULL,
  domains = NULL
)
```

Arguments

object	a prepped object (output of prep()).
file	output path; if NULL, a temporary .html file.
open	logical; open the file in the browser.
plots	logical; add per-step plots (weight before-vs-after scatter and adjustment-factor histogram). Uses ggplot2 if installed, else base graphics.
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 JK _n .
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.

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

*Example survey sample (select-one-person, multistage)***Description**

A realistic multistage design (stratum -> PSU -> household, then one selected person per household). Unknown-eligibility and ineligible addresses appear as single rows with no roster; resolved eligible households are either reached (a roster is obtained) or are household nonresponse; in reached households one person is selected with an unequal within-household probability and may or may not respond. Supports the full household pipeline: household-level eligibility (`cluster`), dropping ineligibles, household and person nonresponse, and `step_select_within`. Generated by `data-raw/weightflow_data.R`.

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	<i>Example survey sample (take-all roster)</i>
---------------	--

Description

A stratified household sample drawn from population where every eligible person in the household is kept (take-all roster). Carries unequal design base weights, an unknown-eligibility flag and a person-level response indicator; survey outcomes (income, employed) are observed only for respondents. Generated by `data-raw/weightflow_data.R`.

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 conditions on the weights at this point of the cascade</i>
-------------	--

Description

A checkpoint that does NOT change the weights; it verifies conditions and fails (error) or warns if they are not met. Useful to guard a production pipeline (tidymodels-style tests inside the recipe).

Usage

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

Arguments

spec	a weighting_spec.
max_deff	numeric or NULL. Maximum acceptable Kish design effect.
max_weight_ratio	numeric or NULL. Maximum allowed final/base weight ratio (per active unit).
min_n_eff	numeric or NULL. Minimum acceptable effective sample size.
on_fail	"error" (stop the cascade) or "warning".

Value

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

Examples

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

step_calibrate	<i>Calibration to population totals</i>
----------------	---

Description

Adjusts the weights so that the weighted sample reproduces known population totals of auxiliary variables, while staying as close as possible to the input weights (Deville & Sarndal 1992). Supports raking (IPF on categorical margins), post-stratification, and linear/GREG calibration, optionally bounded (a logit distance or explicit bounds on the calibration factor). For linear calibration, penalty enables ridge (penalized) calibration, which relaxes the targets to control extreme weights when there are many auxiliaries.

Usage

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

Arguments

spec	a weighting_spec.
margins	named list (classic format for "raking"/"poststratify"). Each element is a named numeric vector with the target totals per category. E.g.: list(sex = c(M = 5000, F = 5200), region = c(N = 3000, S = 7200)). Still fully supported; for a tidy alternative see totals and count.
method	"raking" (IPF, categorical margins), "poststratify" (post-strata: one or more categorical variables crossed) or "linear" (GREG / regression estimator; handles continuous and categorical auxiliaries together).
formula	(only method = "linear") auxiliary formula, e.g. ~ sex + income. Uses model.matrix; includes the intercept unless you write ~ 0 + ...
totals	population totals, in one of two forms. Classic (all methods): for "linear" a named numeric vector aligned with the model.matrix columns (including "(Intercept)" = N); for "raking"/"poststratify" use margins. Tidy (recommended): a data frame or a named list of data frames/numbers giving the totals in a friendly way, paired with count. For "poststratify", a single data frame with one or more category columns plus a counts column. For "raking", a list of data frames, one per margin. For "linear", a named list whose names match the formula terms: a data frame with all categories for each factor, and a single number for each continuous auxiliary; weightflow builds the model.matrix totals internally (you never handle the intercept or dropped reference category).
count	(tidy totals only) string naming the counts column in the totals data frame(s). All other columns are treated as category variables.
by	(tidy totals only) NULL, or a string naming a domain (partition) column. When given, the weights are calibrated independently within each domain , each to its own totals (partitioned / domain calibration). The totals tables carry the domain as a column, and each count table is split by it; a continuous total becomes a data frame domain, value (one total per domain). The domain variable must NOT appear in formula / the margins: it is the partition. Composes with calfun, bounds, penalty and equal_within_cluster, applied within each domain. NULL (default) calibrates globally, as before.
cluster	(only method = "linear") name of the cluster id column (e.g. "household"), for equal weights within the cluster.
equal_within_cluster	(only method = "linear") logical. If TRUE, Lemaitre-Dufour (1987) integrative calibration: a single weight per cluster. Requires cluster. Final weights are equal within the cluster provided the incoming weight is also uniform within the cluster. Works with any calfun distance ("linear", "raking" or "logit"), with bounds and with by.
calfun	(only method = "linear") distance function for the calibration factor g: "linear" (g = 1 + u, closed form), "raking" (g = exp(u), the exponential/multiplicative distance, which keeps the weights positive and still satisfies the constraints exactly) or "logit" (bounded by construction; requires bounds). "raking" and "logit" use the iterative Deville-Sarndal solver and work with the integrative option (equal_within_cluster) too.

bounds	(only method = "linear") numeric c(L, U) with $L < 1 < U$. Bounds on the calibration factor g (g-weights). With "linear" it truncates; with "logit" it is enforced smoothly. Avoids extreme/negative weights without a separate trimming step.
maxit, tol	convergence control for raking and bounded calibration.
penalty	(only method = "linear", unbounded) NULL or positive cost(s) for ridge (penalized) calibration. A positive scalar applies the same cost to every constraint; a named vector sets a cost per constraint (matched to the model.matrix columns). The cost is scale-free: a large value keeps the constraint (near) exact, a small value relaxes it to control extreme weights when there are many auxiliaries. Under ridge the achieved totals no longer match the targets exactly; the diagnostics report the deviation.

Value

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

Examples

```
# Raking to population margins
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_calibrate(method = "raking",
                 margins = list(sex = c(table(population$sex)),
                               region = c(table(population$region)))) |>
  prep()

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

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

# Raking: a list of data frames, one per margin.
m_region <- as.data.frame(table(region = population$region))
m_sex <- as.data.frame(table(sex = population$sex))
```

```

weighting_spec(sample_survey, base_weights = pw) |>
  step_calibrate(method = "raking",
                 totals = list(m_region, m_sex), count = "Freq") |>
  prep()

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

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

```

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

Description

Sets the weight of known-ineligible units to zero so they leave the cascade (excluded from every later step and from collect_weights). No redistribution is done.

Usage

```
step_drop_ineligible(spec, ineligible)
```

Arguments

spec	a weighting_spec.
ineligible	a 0/1 dummy column (1 = ineligible) or any logical condition (unquoted) that is TRUE for out-of-scope units.

Details

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

Value

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

Examples

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

step_model_calibration

Model calibration (model-assisted, Wu & Sitter 2001)

Description

Fits a working model for each study variable y, predicts over the population, and calibrates the weights so that the sample total of each prediction equals its population total (model-assisted efficiency). It also calibrates to the X totals (consistency with the auxiliary controls).

Usage

```
step_model_calibration(
  spec,
  x_formula,
  models,
  population,
  x_totals = NULL,
  count = "Freq",
  cluster = NULL,
  equal_within_cluster = FALSE,
  crossfit = NULL,
  crossfit_seed = NULL
)
```

Arguments

<code>spec</code>	a <code>weighting_spec</code> .
<code>x_formula</code>	formula of the consistency auxiliaries, e.g. <code>~ sex + region</code> .
<code>models</code>	named list of models created with <code>y_model()</code> . The names label the prediction constraints.
<code>population</code>	population data.frame with the auxiliary and predictor columns (the y variables are not needed; they are predicted). Always required: the model-assisted block predicts each y over every population unit, which cannot be done from aggregated totals.
<code>x_totals</code>	optional population totals for the consistency auxiliaries (<code>x_formula</code>), for when they come from an external source rather than from <code>population</code> (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 <code>population</code> . When given, the X totals no longer require <code>x_formula</code> columns to exist in <code>population</code> (only in the sample), and <code>population</code> is used only for the model predictions.
<code>count</code>	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.
<code>cluster</code>	name of the cluster id column (e.g. "household"), for equal weights within the cluster.
<code>equal_within_cluster</code>	logical. If TRUE, integrative calibration: a single weight per cluster. Requires <code>cluster</code> and that the incoming weight be uniform within the cluster.
<code>crossfit</code>	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 <code>cluster</code> when given. NULL (default) fits and predicts in-sample.
<code>crossfit_seed</code>	integer or NULL. Seed for reproducible fold assignment.

Details

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

Value

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

Examples

```
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class", by = "region") |>
  step_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 respondents to represent the nonrespondents, under the assumption that response is ignorable given the information used. The response propensity can be estimated by weighting classes (cells), by a model ("propensity"), with engines ranging from logistic regression to machine learning (regression tree, random forest, gradient boosting), or the adjustment can be made by calibrating the respondents to auxiliary totals ("calibration", the two-phase / Sarndal-Lundstrom approach). Optional K-fold cross-fitting estimates the propensity out-of-sample to avoid the overfitting that flexible engines can introduce. The adjustment can be applied at the person or, via `cluster`, the household level.

Usage

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

Arguments

<code>spec</code>	a <code>weighting_spec</code> .
<code>respondent</code>	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.
<code>method</code>	"weighting_class" (cells), "propensity" (predictive model) or "calibration" (calibrate the respondents to auxiliary totals; two-phase / Sarndal-Lundstrom).
<code>by</code>	character. Adjustment cells for <code>method = "weighting_class"</code> .
<code>formula</code>	predictor formula (right-hand side only), e.g. <code>~ age + region</code> , used when <code>method = "propensity"</code> .
<code>engine</code>	engine to estimate the propensity when <code>method = "propensity"</code> : "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.
cluster	character or NULL. If given, the adjustment is done at the cluster (e.g. household) level for whole-household nonresponse: each household counts once with its (uniform) weight; in "weighting_class" the redistribution is between responding and nonresponding households within the cells, and in "propensity" the model is fitted with one row per household (household auxiliaries), predicting the household response. The resulting factor is assigned to every member; non-responding households 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 household) calibration.
crossfit	integer or NULL. If given (number of folds K >= 2), the propensity is estimated by K-fold cross-fitting: for each fold the model is trained on the other folds and used to predict the held-out fold, so each unit's propensity comes from a model that did not see it. This avoids the overfitting that flexible engines (forest, boost) can produce, which would otherwise inflate the weights. Folds are formed by cluster when given (so correlated units stay together). NULL (default) fits and predicts in-sample.
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.

Value

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

Examples

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

# household-level nonresponse (whole household responds or not)
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "weighting_class",
    by = "region", cluster = "household_id") |>
  prep()

# propensity with cross-fitting (out-of-sample, avoids overfitting)
weighting_spec(sample_survey, base_weights = pw) |>
  step_nonresponse(respondent = responded, method = "propensity",
    formula = ~ region + sex, engine = "logit",
    num_classes = 5, crossfit = 5, crossfit_seed = 1) |>
  prep()

# gradient boosting engine (requires the 'xgboost' package)

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

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

step_rescale	<i>Rescale (normalize) the weights</i>
--------------	--

Description

Rescale (normalize) the weights

Usage

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

Arguments

spec	a weighting_spec.
to	"n" (weights sum to the number of active units, i.e. mean weight 1) or "total" (weights sum to total).
total	numeric. Target sum when to = "total".
by	character. Rescale within these groups (optional). With to = "n", each group sums to its own active count.

Value

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

Examples

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

step_round	<i>Round the final weights</i>
------------	--------------------------------

Description

Optional step, typically the last one (after calibration). Simple rounding ("nearest") slightly breaks the calibrated totals; "preserve_total" uses the largest-remainder method to keep the exact total.

Usage

```
step_round(spec, digits = 0L, method = c("nearest", "preserve_total"))
```

Arguments

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

Value

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

Examples

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

step_select_within	<i>Within-household selection adjustment</i>
--------------------	--

Description

When one (or a subsample) of the eligible persons is selected within each household, the selected person represents all eligible persons, so the weight is multiplied by the inverse of the within-household selection probability. Apply it after the (household-level) eligibility adjustment and before the nonresponse adjustment.

Usage

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

Arguments

spec	a weighting_spec.
prob	unquoted column with the within-household selection probability of the selected person (need not be 1/n_eligible). The weight is multiplied by 1/prob.
n_eligible	unquoted column with the number of eligible persons in the household, for simple random selection within the household. When a single person is selected (the default), the weight is multiplied by n_eligible (equivalent to prob = 1/n_eligible).
n_selected	optional number of persons selected per household under simple random selection, when more than one person is subsampled. Either a single number (same subsample size in every household) or an unquoted column (subsample size varying by household). The weight is multiplied by n_eligible / n_selected (equivalent to prob = n_selected/n_eligible). Defaults to 1. Only used together with n_eligible.

Value

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

Examples

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

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

step_trim	<i>Trim extreme weights</i>
-----------	-----------------------------

Description

Caps weights above a limit and, optionally, redistributes the excess among the others to preserve the weighted total (Potter 1988, 1990; Liu et al. 2004). Optional step that can be inserted anywhere in the recipe, even several times. Operates on the **CURRENT** weights at that point of the cascade.

Usage

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

Arguments

<code>spec</code>	a <code>weighting_spec</code> .
<code>max_ratio</code>	number. Upper cap. Its meaning depends on <code>reference</code> . E.g. with <code>reference = "base"</code> and <code>max_ratio = 4</code> , no weight may exceed 4 times its design weight.
<code>min_ratio</code>	number or <code>NULL</code> . Lower floor (same units as <code>max_ratio</code>).
<code>reference</code>	"base" (multiple of each unit's base weight), "median" (multiple of the median of current weights) or "value" (absolute weight value).

redistribute	logical. If TRUE, redistributes the trimmed excess among the uncapped weights to preserve the total (iterating). If you calibrate afterwards you can use FALSE: calibration restores the totals.
by	character. Groups within which to redistribute (optional).
maxit	integer. Maximum cap+redistribution iterations.

Details

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

Value

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

Examples

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

step_trim_calibrated	<i>Trimmed calibration (range-restricted, totals-preserving)</i>
----------------------	--

Description

Trims already-calibrated weights into an absolute interval `[lower, upper]` **while preserving the calibration totals** of formula. Unlike `step_trim_weights()` (which caps and then redistributes the trimmed mass, so the calibration constraints are broken), this is a bounded re-calibration: it finds the weights closest to the incoming ones that both lie in `[lower, upper]` and still reproduce the totals the incoming weights achieve (the generalized exponential method of Folsom & Singh 2000). The absolute-weight bound is imposed as a per-unit factor bound w_{new} / w in `[lower/w, upper/w]` on top of the incoming weights, using 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.

Usage

```
step_trim_calibrated(
  spec,
  formula,
  lower = NULL,
  upper = NULL,
  calfun = c("linear", "raking"),
  cluster = NULL,
```

```

    equal_within_cluster = FALSE,
    maxit = 100L,
    tol = 1e-07
  )

```

Arguments

spec	a weighting_spec.
formula	the auxiliaries whose calibration totals must be preserved (right-hand side only), e.g. ~ region + age_group. Usually the same formula used in the preceding step_calibrate().
lower, upper	numeric. Absolute bounds on the trimmed weight. At least one must be supplied; the other defaults to no bound. For positive variance, use a positive lower.
calfun	distance function: "linear" (default; the range-restricted Euclidean distance) or "raking" (the multiplicative distance, which keeps the adjustment factors positive).
cluster	character or NULL. Cluster (e.g. household) id column, for integrative trimming (with equal_within_cluster = TRUE).
equal_within_cluster	logical. If TRUE, integrative trimming: one trimming factor per cluster, so weights stay constant within household. The incoming weights must already be constant within cluster (e.g. from step_calibrate(equal_within_cluster = TRUE)); the absolute bound then applies to that common household weight. Requires cluster. FALSE (default) trims each unit on its own.
maxit	integer. Maximum iterations for the bounded solver.
tol	numeric. Convergence tolerance for the bounded solver.

Details

This step is meant to run **after** a step_calibrate(): it acts on the positive incoming weights and leaves dropped units (weight 0) alone.

Value

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

References

Folsom, R. E. and Singh, A. C. (2000). The generalized exponential model for sampling weight calibration for extreme values, nonresponse, and poststratification. *ASA Proceedings of the Section on Survey Research Methods*, 598-603.

Examples

```

# calibrate, then trim the calibrated weights into [50, 400] without breaking
# the region/sex totals
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 = 50, upper = 400) |>
prep()

```

step_trim_weights	<i>Automatic weight trimming (survey-style)</i>
-------------------	---

Description

Caps weights into [lower, upper] and redistributes the change among the untrimmed units to preserve the total. With `redistribute = "uniform"` the change is shared equally among the untrimmed units (and cases already trimmed are never reused), exactly mirroring `survey::trimWeights()`; the default `"proportional"` shares it in proportion to the untrimmed weights, keeping their relative sizes. By default no weight may fall below 1, and the upper cap is chosen by an automatic rule: the Tukey far-out fence ($Q3 + 3 \cdot IQR$) or, with `method = "potter"`, Potter's MSE-optimal cutoff.

Usage

```

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

```

Arguments

<code>spec</code>	a <code>weighting_spec</code> .
<code>lower</code>	numeric. Lower floor (default 1: no weight below 1).
<code>upper</code>	numeric or <code>NULL</code> . Upper cap. If <code>NULL</code> , the cap is chosen automatically by <code>method</code> .
<code>method</code>	rule for the automatic cap when <code>upper = NULL</code> : <code>"tukey"</code> (default, $Q3 + 3 \cdot IQR$ far-out fence) or <code>"potter"</code> (Potter's MSE-optimal cutoff, which over a grid of candidate cutoffs minimizes an estimate of $\text{bias}^2 + \text{variance}$ and so balances the bias of trimming against the variance from extreme weights). Ignored when <code>upper</code> is supplied.
<code>redistribute</code>	how the trimmed mass is shared among the untrimmed units: <code>"proportional"</code> (default; in proportion to their weights, preserving relative sizes) or <code>"uniform"</code> (an equal amount to each untrimmed unit, and units already trimmed are not reused, exactly reproducing <code>survey::trimWeights()</code>).

strict	logical. If TRUE (default), iterate cap+redistribution until no weight is outside [lower, upper] (like survey's strict = TRUE). If FALSE, a single pass (redistribution may push some weights slightly past the cap).
maxit	integer. Maximum iterations when strict = TRUE.

Value

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

Examples

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

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

step_unknown_eligibility

Unknown-eligibility adjustment

Description

Redistributes the weight of unknown-eligibility cases among the known-eligibility cases, within the cells defined by `by`.

Usage

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

Arguments

spec	a <code>weighting_spec</code> .
unknown	a 0/1 dummy column (1 = eligibility unknown) or any logical condition (unquoted) that is TRUE for unknown-eligibility cases. Evaluated on the data.
by	character. Variables defining the adjustment cells (optional).
cluster	character. Cluster (e.g. household) id column. If given, the redistribution is done at the cluster level: each cluster counts once with its (uniform) weight, the weight of unknown-eligibility clusters is redistributed among the known ones, and the adjusted weight is assigned to every member. Use this when unknown-eligibility units have no roster (one row per address) while resolved units are expanded by person.

Value

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

Examples

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

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

```
summary.prepped_weighting_spec
  Detailed per-step diagnostics
```

Description

Detailed per-step diagnostics

Usage

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

Arguments

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

Value

(invisibly) the prepped object.

Examples

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

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

Description

Creates an inert recipe object. Nothing is computed until `prep()` is called.

Usage

```
weighting_spec(data, base_weights)
```

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.

Value

an object of class "weighting_spec".

Examples

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

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

Description

Returns a data.frame with the weight at each stage and the factor of each step (stage weight / previous-stage weight), handy for custom plots.

Usage

```
weight_factors(object)
```

Arguments

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

Value

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

Examples

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

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

Description

Specify a working model for a study variable y

Usage

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

Arguments

<code>formula</code>	full formula, e.g. <code>income ~ sex + age_g</code> .
<code>engine</code>	"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> .
<code>family</code>	for engine = "glm": "gaussian", "binomial" or "poisson". For tree/forest, regression vs classification is inferred from y.

Value

a model specification list.

Examples

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

Index

* datasets

- population, [11](#)
- sample_one, [14](#)
- sample_survey, [15](#)

- as_svrepdesign (as_svydesign), [3](#)
- as_svydesign, [3](#)

- boot_mean (bootstrap_estimate), [3](#)
- boot_total (bootstrap_estimate), [3](#)
- bootstrap_estimate, [3](#)
- bootstrap_weights, [4](#)

- collect_replicate_weights, [5](#)
- collect_weights, [6](#)

- design_effect, [7](#)

- jack_mean (jackknife_estimate), [8](#)
- jack_total (jackknife_estimate), [8](#)
- jackknife_estimate, [8](#)
- jackknife_weights, [9](#)

- plot.prepped_weighting_spec, [10](#)
- population, [11](#)
- prep, [11](#)

- report_weighting, [12](#)

- sample_one, [14](#)
- sample_survey, [15](#)
- step_assert, [15](#)
- step_calibrate, [16](#)
- step_drop_ineligible, [19](#)
- step_model_calibration, [20](#)
- step_nonresponse, [22](#)
- step_rescale, [26](#)
- step_round, [26](#)
- step_select_within, [27](#)
- step_trim, [28](#)
- step_trim_calibrated, [29](#)

- step_trim_weights, [31](#)
- step_unknown_eligibility, [32](#)
- summary.prepped_weighting_spec, [33](#)

- weight_factors, [34](#)
- weighting_spec, [34](#)

- y_model, [35](#)