Package 'WeightIt'

September 30, 2026

Type Package

Title Weighting for Covariate Balance in Observational Studies

Version 2.1.0

Description Generates balancing weights for causal effect estimation in observational studies with binary, multi-category, or continuous point or longitudinal treatments by easing and extending the functionality of several R packages and providing in-house estimation methods. Available methods include those that rely on parametric modeling, optimization, and machine learning. Also
allows for assessment of weights and checking of covariate balance by interfacing directly with the 'cobalt' package. Methods for estimating weighted regression models that take into account
uncertainty in the estimation of the weights via M-estimation or bootstrapping are available. See the vignette ``Installing Supporting Packages'' for instructions on how
to install any optional package 'WeightIt' uses, including those that may not be on CRAN.

Depends R (>= 4.1.0)

Imports cobalt (>= 5.0.0), ggplot2 (>= 4.0.0), cli (>= 3.6.5), rlang (>= 1.1.0), sandwich, generics, utils, stats, arg (>= 0.2.1)

Suggests rootSolve (>= 1.8.2.4), CBPS (>= 0.18), optweight (>= 2.0.1), SuperLearner (>= 2.0-25), mclogit, MNP (>= 3.1-4), brglm2 (>= 0.5.2), enrichwith (>= 0.3.1), logistf (>= 1.26.0), lme4 (>= 1.1-31), osqp (>= 1.0.0), survival (>= 3.6-2), fwb (>= 0.2.0), splines, marginaleffects (>= 0.19.0), adrftools, MASS, gbm (>= 2.1.9), dbarts (>= 0.9-29), stan4bart, misaem (>= 1.0.1), GPBayes, mlogit, dfidx, broom, future, future.apply, knitr, rmarkdown, testthat (>= 3.0.0), waldo (>= 0.6.0)

License GPL (>= 2)

Language en-US

Encoding UTF-8

URL https://ngreifer.github.io/WeightIt/, https://github.com/ngreifer/WeightIt

BugReports https://github.com/ngreifer/WeightIt/issues

VignetteBuilder knitr

LazyData true

Config/testthat/edition 3

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Noah Greifer [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0003-3067-7154>)

Maintainer Noah Greifer <noah.greifer@gmail.com>

Repository CRAN

Date/Publication 2026-09-30 21:21:22 UTC

Contents

.cens . . . 3
.weightit_methods . . . 5
anova.glm_weightit . . . 8
as.weightit . . . 9
calibrate . . . 11
coxph_weightit . . . 13
ESS . . . 15
get_w_from_ps . . . 17
glm_weightit . . . 21
glm_weightit-methods . . . 25
make_full_rank . . . 28
method_bart . . . 29
method_cbps . . . 33
method_cfd . . . 39
method_ebal . . . 43
method_energy . . . 47
method_gbm . . . 52
method_glm . . . 59
method_ipt . . . 66
method_npcbps . . . 69
method_optweight . . . 71
method_ps . . . 75
method_super . . . 77
method_user . . . 82
msmdata . . . 86
multinom_weightit . . . 87
ordinal_weightit . . . 90
plot.weightit . . . 95
predict.glm_weightit . . . 96
sbps . . . 99
summary.weightit . . . 102
trim . . . 104
weightit . . . 107

weightit.fit 112
weightitMSM 116

Index **122**

`.cens` *Mark a censoring indicator in a model formula*

Description

`.cens()` marks a variable as a censoring indicator rather than a treatment, requesting inverse probability of censoring weights (IPCW). It is most often used on the left side of a formula supplied to `weightit()` or in the `formula.list` of `weightitMSM()`, as in `.cens(C) ~ x1 + x2`. It can also be called directly to tag a variable for use with the lower-level interfaces `weightit.fit()` and `get_w_from_ps()`.

This is **cobalt**'s `cobalt::.cens()`, re-exported. **cobalt** defines it because `cobalt::bal.tab()` needs the same marker to assess a censoring model's balance; there is one function, so an indicator tagged with either package attached is the same object and is accepted by both.

Usage

```
.cens(x)
```

Arguments

`x`	a censoring indicator: a numeric variable taking only the values 0 (still under observation) and 1 (censored), a logical variable, or a factor with levels `0`/`1` or `FALSE`/`TRUE`.

Details

Censoring is treated as its own treatment type, distinct from binary, multi-category, and continuous treatments. Weights are estimated only for the units still under observation, and are those that make the covariate distribution of the uncensored units match that of the full at-risk sample. Formally, for $e(X) = P(C = 1|X)$, the weights are $(1 - e(X))^{-1}$ for units with $C = 0$ and exactly 0 for units with $C = 1$.

Because only one group is weighted, the estimation problem is smaller and better conditioned than the corresponding binary-treatment problem, which would additionally solve for weights among the censored units. This matters most when few units are censored.

Because censored units receive a weight of exactly 0, they contribute nothing to a weighted outcome model, and `glm_weightit()`, `lm_weightit()`, `multinom_weightit()`, `ordinal_weightit()`, and `coxph_weightit()` all tolerate missing values in the model variables for those units. This is what makes it possible to fit an outcome model whose outcome is unobserved after censoring. See the *Censoring weights (IPCW)* section of `weightit()` for details.

No `estimand` or `focal` argument applies to censoring models; supplying one produces a warning and it is ignored. `subclass` cannot be used. Not all methods support censoring weights; see the `treat_type` component of `.weightit_methods` to check.

The right side of the formula may be empty, as in `.cens(C) ~ 1`, requesting a marginal censoring model in which censoring is assumed independent of the covariates. The weights are then $1/P(C = 0)$ for the units still under observation and 0 for the censored units, whatever `method` is supplied; see *Empty model formulas* in Details at `weightit()`. Nothing else changes, so a marginal censoring model still composes with by, `stabilize`, M-estimation, and, in `weightitMSM()`, the risk sets and the missing values permitted after censoring.

Use outside a formula:

Calling `.cens()` directly returns a tagged indicator that routes the lower-level interfaces to the censoring path:

```
# Equivalent to weightit(.cens(C) ~ x1 + x2, ...)
weightit.fit(covs, treat = .cens(C), method = "glm")

# Censoring weights from a set of censoring probabilities
get_w_from_ps(ps, treat = .cens(C))
```

Tagging is the only way to request censoring weights from these interfaces; an untagged 0/1 vector is treated as an ordinary binary treatment.

Value

`x` coerced to a 0/1 numeric vector of class `treat` (see cobalt::treat) with a `"treat.type"` attribute of `"censoring"`. Missing values are preserved; any value other than 0 or 1 throws an error.

Inside a formula the marker is stripped before the formula is processed, so `.cens()` is not actually evaluated there and the treatment name remains that of the indicator itself (e.g., `C` rather than `.cens(C)`).

Note on the coding convention

The survival convention is used: `1` means the unit is *censored* (drops out of observation) and `0` means it remains under observation. This is the opposite of an "observed" or "event" indicator. When a censoring model is supplied to `weightit()`, the returned propensity score is $P(C = 1|X)$, the probability of *being censored*.

Assessing balance

Because the target of a censoring model is the full at-risk sample rather than another treatment group, balance is assessed by comparing the weighted uncensored units against that sample.

`cobalt::bal.tab()` does this directly: it accepts a `weightit` object fit with a censoring model, as well as a `weightitMSM` object with censoring among its time points, and produces the comparison described above for each. See `cobalt::class-bal.tab.cens`.

The same quantity can also be built by hand, by stacking the two samples into a binary pseudo-treatment:

```
u <- which(W$treat == 0)
bal.tab(rbind(covs[u, ], covs),
        treat = c(rep(0L, length(u)), rep(1L, nrow(covs))),
        weights = c(W$weights[u], rep(1, nrow(covs))),
        estimand = "ATT", s.d.denom = "treated")
```

The "Control" group is then the weighted uncensored sample and the "Treated" group the full at-risk sample, so `Diff.Adj` is the quantity the weights are designed to zero out. For a `weightitMSM` object, the `at.risk` component, which has one column per time point, restricts each model to the units that were under observation when it was fit:

```
ar <- W$at.risk[, "A_3"]
bal.tab(A_3 ~ X1_2 + X2_2 + A_2, data = data[ar, ],
        weights = W$weights[ar], s.weights = W$s.weights[ar])
```

See Also

`weightit()` and `weightitMSM()` for estimating censoring weights; `.weightit_methods` for which methods support them; `cobalt::.cens()` for the function itself and `cobalt::class-bal.tab.cens` for the balance output.

Examples

```
data("msmdata")

# A censoring indicator: 1 = lost to follow-up after time 2
set.seed(1234)
msmdata$C_2 <- rbinom(nrow(msmdata), 1,
                      prob = plogis(-2 + 0.2 * msmdata$X1_1))

# Inverse probability of censoring weights
(Wc <- weightit(.cens(C_2) ~ X1_1 + X2_1 + A_1,
                data = msmdata, method = "glm"))

# Censored units have a weight of 0
all(Wc$weights[msmdata$C_2 == 1] == 0)

# The weights are the inverse probability of remaining
# under observation
all.equal(Wc$weights[msmdata$C_2 == 0],
          1 / (1 - Wc$ps[msmdata$C_2 == 0]))

# `.cens()` can also tag a variable directly for use with
# the lower-level interfaces
C <- .cens(msmdata$C_2)

all.equal(get_w_from_ps(Wc$ps, treat = C),
          Wc$weights, check.attributes = FALSE)
```

`.weightit_methods` *Weighting methods*

Description

`.weightit_methods` is a list containing the allowable weighting methods that can be supplied by name to the `method` argument of `weightit()`, `weightitMSM()`, and `weightit.fit()`. Each entry corresponds to an allowed method and contains information about what options are and are not allowed for each method. While this list is primarily for internal use by checking functions in **WeightIt**, it might be of use for package authors that want to support different weighting methods.

Usage

```
.weightit_methods
```

Format

A named list with one component per weighting method, named for the method's canonical name (the value accepted by `method`). Each component is itself a named list of the components described in Details.

Details

Each component is itself a list containing the following components:

`treat_type` at least one of `"binary"`, `"multinomial"`, or `"continuous"` indicating which treatment types are available for this method. `"censoring"` indicates the method can also estimate inverse probability of censoring weights when the censoring indicator is wrapped in `.cens()`.

`estimand` which estimands are available for this method. Most methods that support binary and multi-category treatments accept `"ATE"`, `"ATT"`, and `"ATC"`, as well as some other estimands depending on the method. See `get_w_from_ps()` for more details about what each estimand means.

`alias` a character vector of aliases for the method. When an alias is supplied, the corresponding method will still be dispatched. For example, the canonical method to request entropy balancing is `"ebal"`, but `"ebalance"` and `"entropy"` also work. The first value is the canonical name.

`description` a string containing the description of the name in English.

`re_ok` a logical for whether random effects terms (e.g., `(1 | group)`) can be included in the model `formula` supplied to `weightit()` to fit a multilevel version of the method.

`ps` a logical for whether propensity scores are returned by the method for binary treatments. Propensity scores are never returned for multi-category or continuous treatments.

`msm_valid` a logical for whether the method can be validly used with longitudinal treatments.

`msm_method_available` a logical for whether a version of the method can be used that estimates weights using a single model rather than multiplying the weights across time points. This is related to the `is.MSM.method` argument of `weightitMSM()`.

`subclass_ok` a logical for whether `subclass` can be supplied to compute subclassification weights from the propensity scores.

`packages_needed` a character vector of the minimal packages required to use the method. Some methods may require additional packages (not listed) for certain options.

`package_versions_needed` a named character vector of the minimal package versions required to use the method. Only present when `packages_needed` is not empty.

`s.weights_ok` a logical for whether sampling weights can be used with the method.

`missing` a character vector of the allowed options that can be supplied to `missing` when missing data is present. All methods accept `"ind"` for the missingness indicator approach; some other methods accept additional values.

`moments_int_ok` a logical for whether `moments`, `int`, and `quantile` can be used with the method.

`moments_default` when `moments_int_ok` is `TRUE`, the default value of `moments` used with the method. For most methods, this is 1.

`density_ok` a logical for whether arguments that control the density can be used with the method when used with a continuous treatment.

`stabilize_ok` a logical for whether the `stabilize` argument (and `num.formula` for longitudinal treatments) can be used with the method.

`plot.weightit_ok` a logical for whether `plot()` can be used on the `weightit` output with the method.

See Also

`weightit()` and `weightitMSM()` for how the methods are used. Also see the individual methods pages for information on whether and how each option can be used.

Examples

```
# Get all acceptable names
names(.weightit_methods)

# Get all acceptable names and aliases
lapply(.weightit_methods, `[[`, "alias")

# Which estimands are allowed with `method = "bart"`
.weightit_methods[["bart"]]$estimand

# All methods that support continuous treatments
supp <- sapply(.weightit_methods, function(x) {
  "continuous" %in% x$treat_type
})
names(.weightit_methods)[supp]

# All methods that return propensity scores (for
# binary treatments only)
supp <- sapply(.weightit_methods, `[[`, "ps")
names(.weightit_methods)[supp]
```

anova.glm_weightit *Methods for* glm_weightit *objects*

Description

anova() is used to compare nested models fit with glm_weightit(), multinom_weightit(), ordinal_weightit(), or coxph_weightit() using a Wald test that incorporates uncertainty in estimating the weights (if any).

Usage

```
## S3 method for class 'glm_weightit'
anova(
  object,
  object2,
  test = "Chisq",
  method = "Wald",
  tolerance = 1e-07,
  vcov = NULL,
  ...
)
```

Arguments

object, object2	an output from one of the above modeling functions. object2 is required.
test	the type of test statistic used to compare models. Currently only "Chisq" (the chi-square statistic) is allowed.
method	the kind of test used to compare models. Currently only "Wald" is allowed.
tolerance	for the Wald test, the tolerance used to determine if models are symbolically nested.
vcov	either a string indicating the method used to compute the variance of the estimated parameters for object, a function used to extract the variance, or the variance matrix itself. Default is to use the variance matrix already present in object. If a string or function, arguments passed to ... are supplied to the method or function. (Note: for vcov(), can also be supplied as type.)
...	other arguments passed to the function used for computing the parameter variance matrix, if supplied as a string or function, e.g., cluster, R, or fwb.args.

Details

anova() performs a Wald test to compare two fitted models. The models must be nested, but they don't have to be nested symbolically (i.e., the names of the coefficients of the smaller model do not have to be a subset of the names of the coefficients of the larger model). The larger model must be supplied to object and the smaller to object2. Both models must contain the same units, weights (if any), and outcomes. The variance-covariance matrix of the coefficients of the smaller model is not used.

Value

An object of class "anova" inheriting from class "data.frame".

See Also

glm_weightit() for the page documenting glm_weightit(), lm_weightit(), ordinal_weightit(), multinom_weightit(), and coxph_weightit(). anova.glm() for model comparison of glm objects.

Examples

```
data("lalonde", package = "cobalt")

# Model comparison for any relationship between `treat`
# and `re78` (not the same as testing for the ATE)
fit1 <- glm_weightit(
  re78 ~ treat * (age + educ + race + married + nodegree +
                    re74 + re75), data = lalonde
)

fit2 <- glm_weightit(
  re78 ~ age + educ + race + married + nodegree +
    re74 + re75, data = lalonde
)

anova(fit1, fit2)

# Using the usual maximum likelihood variance matrix
anova(fit1, fit2, vcov = "const")

# Using a bootstrapped variance matrix
anova(fit1, fit2, vcov = "BS", R = 100)


# Model comparison between spline model and linear
# model; note they are nested but not symbolically
# nested
fit_s <- glm_weightit(re78 ~ splines::ns(age, df =4),
                      data = lalonde)

fit_l <- glm_weightit(re78 ~ age,
                      data = lalonde)

anova(fit_s, fit_l)
```

---

as.weightit *Create a* weightit *object manually*

Description

This function allows users to get the benefits of a `weightit` object when using weights not estimated with `weightit()` or `weightitMSM()`. These benefits include diagnostics, plots, and direct compatibility with **cobalt** for assessing balance.

Usage

```
as.weightit(x, ...)

## S3 method for class 'weightit.fit'
as.weightit(x, covs = NULL, ...)

## Default S3 method:
as.weightit(
  x,
  treat,
  covs = NULL,
  estimand = NULL,
  s.weights = NULL,
  ps = NULL,
  ...
)

as.weightitMSM(x, ...)

## Default S3 method:
as.weightitMSM(
  x,
  treat.list,
  covs.list = NULL,
  estimand = NULL,
  s.weights = NULL,
  ps.list = NULL,
  ...
)
```

Arguments

`x`	required; a `numeric` vector of weights, one for each unit, or a `weightit.fit` object from `weightit.fit()`.
`...`	additional arguments. These must be named. They will be included in the output object.
`covs`	an optional data frame of covariates. For using **WeightIt** functions, this is not necessary, but for use with **cobalt** it is. Note that when using with a `weightit.fit` object, this should not be the matrix supplied to the `covs` argument of `weightit.fit()` unless there are no factor/character variables in it. Ideally this is the original, unprocessed covariate data frame with factor variables included.

treat	a vector of treatment statuses, one for each unit. Required when x is a vector of weights.
estimand	an optional character of length 1 giving the estimand. The text is not checked.
s.weights	an optional numeric vector of sampling weights, one for each unit.
ps	an optional numeric vector of propensity scores, one for each unit.
treat.list	a list of treatment statuses at each time point.
covs.list	an optional list of data frames of covariates at each time point. For using **WeightIt** functions, this is not necessary, but for use with **cobalt** it is.
ps.list	an optional list of numeric vectors of propensity scores at each time point.

Value

An object of class weightit (for as.weightit()) or weightitMSM (for as.weightitMSM()).

Examples

```
treat <- rbinom(500, 1, .3)
weights <- rchisq(500, df = 2)

W <- as.weightit(weights, treat = treat, estimand = "ATE")
summary(W)

# See ?weightit.fit for using as.weightit() with a
# weightit.fit object.
```

---

calibrate *Calibrate Propensity Score Weights*

Description

calibrate() calibrates propensity scores used in weights. This involves fitting a new propensity score model using logistic or isotonic regression with the previously estimated propensity score as the sole predictor. Weights are computed using this new propensity score.

Usage

```
calibrate(x, ...)

## Default S3 method:
calibrate(x, treat, s.weights = NULL, data = NULL, method = "platt", ...)

## S3 method for class 'weightit'
calibrate(x, method = "platt", ...)
```

Arguments

x	a weightit object or a vector of propensity scores. Only binary treatments are supported.
...	not used.
treat	a vector of treatment status for each unit. Only binary treatments are supported.
s.weights	a vector of sampling weights or the name of a variable in data that contains sampling weights.
data	an optional data frame containing the variable named in s.weights when supplied as a string.
method	character; the method of calibration used. Allowable options include "platt" (default) for Platt scaling as described by Gutman et al. (2024) and "isoreg" for isotonic regression as described by van der Laan et al. (2024).

Value

If the input is a weightit object, the output will be a weightit object with the propensity scores replaced with the calibrated propensity scores and the weights replaced by weights computed from the calibrated propensity scores.

If the input is a numeric vector of propensity scores, the output will be a numeric vector of the calibrated propensity scores.

References

Gutman, R., Karavani, E., & Shimoni, Y. (2024). Improving Inverse Probability Weighting by Post-calibrating Its Propensity Scores. *Epidemiology*, 35(4). doi:10.1097/EDE.0000000000001733

van der Laan, L., Lin, Z., Carone, M., & Luedtke, A. (2024). Stabilized Inverse Probability Weighting via Isotonic Calibration. arXiv. https://arxiv.org/abs/2411.06342

See Also

weightit(), weightitMSM()

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

#Using GBM to estimate weights
(W <- weightit(treat ~ age + educ + married +
                 nodegree + re74, data = lalonde,
               method = "gbm", estimand = "ATT",
               criterion = "smd.max"))
summary(W)

#Calibrating the GBM propensity scores
Wc <- calibrate(W)

#Calibrating propensity scores directly
```

```
PSc <- calibrate(W$ps, treat = lalonde$treat)
```

---

coxph_weightit *Fitting (Weighted) Cox Proportional Hazards Models*

Description

coxph_weightit() fits a Cox proportional hazards model with a covariance matrix that accounts for estimation of weights, if supplied, and is a wrapper for functions in the **survival** package. By default, this function uses M-estimation to construct a robust covariance matrix using the estimating equations for the weighting model and the outcome model when available.

Usage

```
coxph_weightit(
  formula,
  data,
  weightit = NULL,
  vcov = NULL,
  cluster,
  R = 500L,
  control = list(...),
  x = FALSE,
  y = TRUE,
  fwb.args = list(),
  ...
)
```

Arguments

formula	an object of class formula (or one that can be coerced to that class): a symbolic description of the model to be fitted. Should include a Surv() term as the response. See survival::coxph() for how this should be specified.
data	a data frame containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which the function is called.
weightit	a weightit or weightitMSM object; the output of a call to weightit() or weightitMSM(). If not supplied, an unweighted model will be fit.
vcov	string; the method used to compute the variance of the estimated parameters. Allowable options include "asympt", which uses the asymptotically correct M-estimation-based method that accounts for estimation of the weights when available; "const", which uses the usual maximum likelihood estimates (only available when weightit is not supplied); "HC0", which computes the robust sandwich variance treating weights (if supplied) as fixed; "BS", which uses the traditional bootstrap (including re-estimation of the weights, if supplied); "FWB",

which uses the fractional weighted bootstrap as implemented in `fwb::fwb()` (including re-estimation of the weights, if supplied); and `"none"` to omit calculation of a variance matrix. If `NULL` (the default), will use `"asympt"` if `weightit` is supplied and M-estimation is available and `"HC0"` otherwise. See the `vcov_type` component of the outcome object to see which was used.

`cluster` optional; for computing a cluster-robust variance matrix, a variable indicating the clustering of observations, a list (or data frame) thereof, or a one-sided formula specifying which variable(s) from the fitted model should be used. Note the cluster-robust variance matrix uses a correction for small samples, as is done in `sandwich::vcovCL()` by default. Cluster-robust variance calculations are available only when `vcov` is `"asympt"`, `"HC0"`, `"BS"`, or `"FWB"`.

`R` the number of bootstrap replications when `vcov` is `"BS"` or `"FWB"`. Default is 500. Ignored otherwise.

`control` a list of parameters for controlling the fitting process, passed to `survival::coxph.control()`.

`x, y` logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value.

`fwb.args` an optional list of further arguments to supply to `fwb::fwb()` when `vcov = "FWB"`.

`...` other arguments passed to `survival::coxph.control()`.

Details

`coxph_weightit()` is essentially a simplified version of `survival::coxph()` to fit weighted survival models that optionally computes a coefficient variance matrix that can be adjusted to account for estimation of the weights if a `weightit` or `weightitMSM` object is supplied to the `weightit` argument. It differs from `coxph()` in a few ways:

- the `cluster` argument (if used) should be specified as a one-sided formula (which can include multiple clustering variables) and uses a small sample correction for cluster variance estimates when specified
- Special formula components, such as `strata()`, `cluster()`, `pspline()`, `frailty()`, `ridge()`, and `tt()` are not allowed
- Only right censoring is allowed, and only two-state models are allowed (i.e., the `Surv()` component of `formula` must be of the form `Surv(time, event)`)
- Time-varying predictors are not allowed and there must be one observation per unit (and the `id` and `istate` arguments to `coxph()` are ignored)
- Weights of 0 are allowed (`coxph()` rejects them). Such units are omitted from the model fit, which is exact rather than an approximation: a unit with a weight of 0 contributes neither its own term nor anything to any risk set denominator, both of which are weighted by the same weights. Missing values in the model variables are tolerated for these units, which makes it possible to fit an outcome model after censoring, where the event time is unascertained for censored units (see `.cens()`). Missing values in units with a nonzero weight produce an error.

When no argument is supplied to `weightit` or there is no `"Mparts"` attribute in the supplied object, the default variance matrix returned will be the "HC0" sandwich variance matrix, which is robust to misspecification of the outcome family (including heteroscedasticity). Otherwise, the default variance matrix uses M-estimation to additionally adjust for estimation of the weights. When

possible, this often yields smaller (and more accurate) standard errors. See the individual methods pages to see whether and when an "Mparts" attribute is included in the supplied object. To request that a variance matrix be computed that doesn't account for estimation of the weights even when a compatible weightit object is supplied, set vcov = "HC0", which treats the weights as fixed.

Bootstrapping can also be used to compute the coefficient variance matrix; when vcov = "BS" or vcov = "FWB", which implement the traditional resampling-based and fractional weighted bootstrap, respectively, the entire process of estimating the weights and fitting the outcome model is repeated in bootstrap samples (if a weightit object is supplied). This accounts for estimation of the weights and can be used with any weighting method. It is important to set a seed using set.seed() to ensure replicability of the results. The fractional weighted bootstrap is more reliable but requires the weighting method to accept sampling weights (which most do, and you'll get an error if it doesn't). Setting vcov = "FWB" and supplying fwb.args = list(wtype = "multinom") also performs the resampling-based bootstrap but with the additional features **fwb** provides (e.g., a progress bar and parallelization).

Value

A coxph_weightit object, which inherits from coxph. See survival::coxph() for details.

Unless vcov = "none", the vcov component contains the covariance matrix adjusted for the estimation of the weights if requested and a compatible weightit object was supplied. The vcov_type component contains the type of variance matrix requested. If cluster is supplied, it will be stored in the "cluster" attribute of the output object, even if not used.

The model component of the output object (also the model.frame() output) will include two extra columns when weightit is supplied: (weights) containing the weights used in the model (the product of the estimated weights and the sampling weights, if any) and (s.weights) containing the sampling weights, which will be 1 if s.weights is not supplied in the original weightit() call.

See Also

- survival::coxph() for fitting Cox proportional hazards models without adjusting standard errors for estimation of the weights.
- glm_weightit() for fitting generalized linear models that adjust for estimation of the weights.
- ordinal_weightit() and multinom_weightit() for fitting ordinal and multinomial regression models that adjust for estimation of the weights.

Examples

```
# See `vignette("estimating-effects")` for an example
```

ESS *Compute effective sample size of weighted sample*

Description

Computes the effective sample size (ESS) of a weighted sample, which represents the size of an unweighted sample with approximately the same amount of precision as the weighted sample under consideration.

Usage

```
ESS(w)
```

Arguments

w	a vector of weights.

Details

The ESS is calculated as $(\sum w)^2 / \sum w^2$. It is invariant to multiplicative scaling of the weights (i.e., multiplying all weights by a nonzero scalar).

Value

A single number, the effective sample size. For non-negative weights it lies between 1 and `length(w)`, and equals `length(w)` only when all the weights are equal. It is `NA` if any weight is missing.

References

McCaffrey, D. F., Ridgeway, G., & Morral, A. R. (2004). Propensity Score Estimation With Boosted Regression for Evaluating Causal Effects in Observational Studies. *Psychological Methods*, 9(4), 403–425. doi:10.1037/1082989X.9.4.403

Shook-Sa, B. E., & Hudgens, M. G. (2020). Power and sample size for observational studies of point exposure effects. *Biometrics*, biom.13405. doi:10.1111/biom.13405

See Also

`summary.weightit()`

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "glm", estimand = "ATE"))

summary(W1)

ESS(W1$weights[W1$treat == 0])
ESS(W1$weights[W1$treat == 1])
```

get_w_from_ps *Compute weights from propensity scores*

Description

Given a vector or matrix of propensity scores, outputs a vector of weights that target the provided estimand.

Usage

```
get_w_from_ps(
  ps,
  treat,
  estimand = "ATE",
  focal = NULL,
  treated = NULL,
  subclass = NULL,
  stabilize = FALSE
)
```

Arguments

ps	a vector, matrix, or data frame of propensity scores. See Details. When treat is a censoring indicator, must be a vector of the probability of *being censored*.
treat	a vector of treatment status for each individual. See Details. Can also be a censoring indicator created with .cens(), in which case inverse probability of censoring weights are returned and the remaining arguments do not apply.
estimand	the desired estimand that the weights should target. Current options include "ATE" (average treatment effect), "ATT" (average treatment effect on the treated), "ATC" (average treatment effect on the control), "ATO" (average treatment effect in the overlap), "ATM" (average treatment effect in the matched sample), and "ATOS" (average treatment effect in the optimal subset). See Details.
focal	when estimand is "ATT" or "ATC", which group should be considered the (focal) "treated" or "control" group, respectively. If not NULL and estimand is not "ATT" or "ATC", estimand will automatically be set to "ATT".
treated	when treatment is binary, the value of treat that is considered the "treated" group (i.e., the group for which the propensity scores are the probability of being in). If NULL, get_w_from_ps() will attempt to figure it out on its own using some heuristics. This really only matters when treat has values other than 0 and 1 and when ps is given as a vector or an unnamed single-column matrix or data frame.
subclass	numeric; the number of subclasses to use when computing weights using marginal mean weighting through stratification (MMWS; also known as fine stratification). If NULL, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and weights

will be computed based on subclass membership. `estimand` must be `"ATE"`, `"ATT"`, or `"ATC"` if `subclass` is non-`NULL`. See Details.

`stabilize` `logical`; whether to compute stabilized weights or not. This simply involves multiplying each unit's weight by the proportion of units in their treatment group. For saturated outcome models and in balance checking, this won't make a difference; otherwise, this can improve performance.

Details

`get_w_from_ps()` applies the formula for computing weights from propensity scores for the desired estimand. The formula for each estimand is below, with A_i the treatment value for unit i taking on values $\mathcal{A} = (1, \dots, g)$, $p_{a,i}$ the probability of receiving treatment level a for unit i, and f is the focal group (the treated group for the ATT and the control group for the ATC):

$$
\begin{aligned}
w_i^{ATE} &= 1/p_{A_i,i} \\
w_i^{ATT} &= w_i^{ATE} \times p_{f,i} \\
w_i^{ATO} &= w_i^{ATE} / \sum_{a \in \mathcal{A}} 1/p_{a,i} \\
w_i^{ATM} &= w_i^{ATE} \times \min(p_{1,i}, \dots, p_{g,i}) \\
w_i^{ATOS} &= w_i^{ATE} \times \mathbb{1}(\alpha < p_{2,i} < 1 - \alpha)
\end{aligned}
$$

`get_w_from_ps()` can only be used with binary and multi-category treatments.

Censoring indicators:

When `treat` is a censoring indicator created with `.cens()`, `get_w_from_ps()` instead returns inverse probability of censoring weights: $(1 - e(X))^{-1}$ for the units still under observation and exactly 0 for the censored units, where `ps` is $e(X) = P(C = 1|X)$, the probability of *being censored*. `estimand`, `focal`, `treated`, `subclass`, and `stabilize` do not apply and are ignored with a warning. See `.cens()`.

Supplying the `ps` argument:

The `ps` argument can be entered in two ways:

- A numeric matrix with a row for each unit and a (named) column for each treatment level, with each cell corresponding to the probability of receiving the corresponding treatment level
- A numeric vector with a value for each unit corresponding to the probability of being "treated" (only allowed for binary treatments)

When supplied as a vector, `get_w_from_ps()` has to know which value of `treat` corresponds to the "treated" group. For 0/1 variables, 1 will be considered treated. For other types of variables, `get_w_from_ps()` will try to figure it out using heuristics, but it's safer to supply an argument to `treated`. When `estimand` is `"ATT"` or `"ATC"`, supplying a value to `focal` is sufficient (for ATT, `focal` is the treated group, and for ATC, `focal` is the control group).

When supplied as a matrix, the columns must be named with the levels of the treatment, and it is assumed that each column corresponds to the probability of being in that treatment group. This is the safest way to supply `ps` unless `treat` is a 0/1 variable. When `estimand` is `"ATT"` or `"ATC"`, a value for `focal` must be specified.

Marginal mean weighting through stratification (MMWS):

When `subclass` is not `NULL`, MMWS weights are computed. The implementation differs slightly from that described in Hong (2010, 2012). First, subclasses are formed by finding the quantiles of the propensity scores in the target group (for the ATE, all units; for the ATT or ATC, just the units in the focal group). Any subclasses lacking members of a treatment group will be filled in with them from neighboring subclasses so each subclass will always have at least one member of each treatment group. A new subclass-propensity score matrix is formed, where each unit's subclass-propensity score for each treatment value is computed as the proportion of units with that treatment value in the unit's subclass. For example, if a subclass had 10 treated units and 90 control units in it, the subclass-propensity score for being treated would be .1 and the subclass-propensity score for being control would be .9 for all units in the subclass.

For multi-category treatments, the propensity scores for each treatment are stratified separately as described in Hong (2012); for binary treatments, only one set of propensity scores is stratified and the subclass-propensity scores for the other treatment are computed as the complement of the propensity scores for the stratified treatment.

After the subclass-propensity scores have been computed, the standard propensity score weighting formulas are used to compute the unstabilized MMWS weights. To estimate MMWS weights equivalent to those described in Hong (2010, 2012), `stabilize` must be set to `TRUE`, but, as with standard propensity score weights, this is optional. Note that MMWS weights are also known as fine stratification weights and described by Desai et al. (2017).

Value

A vector of weights. When `subclass` is not `NULL`, the subclasses are returned as the `"subclass"` attribute. When `estimand = "ATOS"`, the chosen value of `alpha` (the smallest propensity score allowed to remain in the sample) is returned in the `"alpha"` attribute.

References

Binary treatments:

- `estimand = "ATO"`

Li, F., Morgan, K. L., & Zaslavsky, A. M. (2018). Balancing covariates via propensity score weighting. Journal of the American Statistical Association, 113(521), 390–400. doi:10.1080/01621459.2016.1260466

- `estimand = "ATM"`

Li, L., & Greene, T. (2013). A Weighting Analogue to Pair Matching in Propensity Score Analysis. The International Journal of Biostatistics, 9(2). doi:10.1515/ijb20120030

- `estimand = "ATOS"`

Crump, R. K., Hotz, V. J., Imbens, G. W., & Mitnik, O. A. (2009). Dealing with limited overlap in estimation of average treatment effects. Biometrika, 96(1), 187–199. doi:10.1093/biomet/asn055

- Other estimands

Austin, P. C. (2011). An Introduction to Propensity Score Methods for Reducing the Effects of Confounding in Observational Studies. Multivariate Behavioral Research, 46(3), 399–424. doi:10.1080/00273171.2011.568786

- Marginal mean weighting through stratification (MMWS)

Hong, G. (2010). Marginal mean weighting through stratification: Adjustment for selection bias in multilevel data. Journal of Educational and Behavioral Statistics, 35(5), 499–531. doi:10.3102/1076998609359785

Desai, R. J., Rothman, K. J., Bateman, B. . T., Hernandez-Diaz, S., & Huybrechts, K. F. (2017). A Propensity-score-based Fine Stratification Approach for Confounding Adjustment When Exposure Is Infrequent: Epidemiology, 28(2), 249–257. doi:10.1097/EDE.0000000000000595

Multi-Category Treatments:

- `estimand = "ATO"`

Li, F., & Li, F. (2019). Propensity score weighting for causal inference with multiple treatments. The Annals of Applied Statistics, 13(4), 2389–2415. doi:10.1214/19AOAS1282

- `estimand = "ATM"`

Yoshida, K., Hernández-Díaz, S., Solomon, D. H., Jackson, J. W., Gagne, J. J., Glynn, R. J., & Franklin, J. M. (2017). Matching weights to simultaneously compare three treatment groups: Comparison to three-way matching. Epidemiology (Cambridge, Mass.), 28(3), 387–395. doi:10.1097/EDE.0000000000000627

- Other estimands

McCaffrey, D. F., Griffin, B. A., Almirall, D., Slaughter, M. E., Ramchand, R., & Burgette, L. F. (2013). A Tutorial on Propensity Score Estimation for Multiple Treatments Using Generalized Boosted Models. Statistics in Medicine, 32(19), 3388–3414. doi:10.1002/sim.5753

- Marginal mean weighting through stratification (MMWS)

Hong, G. (2012). Marginal mean weighting through stratification: A generalized method for evaluating multivalued and multiple treatments with nonexperimental data. *Psychological Methods*, 17(1), 44–60. doi:10.1037/a0024918

See Also

method_glm

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

ps.fit <- glm(treat ~ age + educ + race + married +
                nodegree + re74 + re75, data = lalonde,
              family = binomial)
ps <- ps.fit$fitted.values

w1 <- get_w_from_ps(ps, treat = lalonde$treat,
                    estimand = "ATT")

treatAB <- factor(ifelse(lalonde$treat == 1, "A", "B"))
w2 <- get_w_from_ps(ps, treat = treatAB,
                    estimand = "ATT", focal = "A")
all.equal(w1, w2)
w3 <- get_w_from_ps(ps, treat = treatAB,
                    estimand = "ATT", treated = "A")
```

```
all.equal(w1, w3)

# Using MMWS
w4 <- get_w_from_ps(ps, treat = lalonde$treat,
                    estimand = "ATE", subclass = 20,
                    stabilize = TRUE)

# A multi-category example using predicted probabilities
# from multinomial logistic regression
T3 <- factor(sample(c("A", "B", "C"), nrow(lalonde),
                    replace = TRUE))

multi.fit <- multinom_weightit(
  T3 ~ age + educ + race + married +
    nodegree + re74 + re75, data = lalonde,
  vcov = "none"
)

ps.multi <- fitted(multi.fit)
head(ps.multi)

w5 <- get_w_from_ps(ps.multi, treat = T3,
                    estimand = "ATE")
```

`glm_weightit` *Fitting (Weighted) Generalized Linear Models*

Description

`glm_weightit()` is used to fit generalized linear models with a covariance matrix that accounts for estimation of weights, if supplied. `lm_weightit()` is a wrapper for `glm_weightit()` with the Gaussian family and identity link (i.e., a linear model). By default, these functions use M-estimation to construct a robust covariance matrix using the estimating equations for the weighting model and the outcome model when available.

Usage

```
glm_weightit(
  formula,
  data,
  family = gaussian,
  weightit = NULL,
  vcov = NULL,
  cluster,
  R = 500L,
  offset,
  start = NULL,
  etastart,
  mustart,
```

```
  control = list(...),
  x = FALSE,
  y = TRUE,
  contrasts = NULL,
  fwb.args = list(),
  br = FALSE,
  ...
)

lm_weightit(
  formula,
  data,
  weightit = NULL,
  vcov = NULL,
  cluster,
  R = 500L,
  offset,
  x = FALSE,
  y = TRUE,
  contrasts = NULL,
  fwb.args = list(),
  ...
)
```

Arguments

formula	an object of class formula (or one that can be coerced to that class): a symbolic description of the model to be fitted.
data	a data frame containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which the function is called.
family	a description of the error distribution and link function to be used in the model. This can be a character string naming a family function, a family function or the result of a call to a family function. See family for details of family functions.
weightit	a weightit or weightitMSM object; the output of a call to weightit() or weightitMSM(). If not supplied, an unweighted model will be fit.
vcov	string; the method used to compute the variance of the estimated parameters. Allowable options include "asympt", which uses the asymptotically correct M-estimation-based method that accounts for estimation of the weights when available; "const", which uses the usual maximum likelihood estimates (only available when weightit is not supplied); "HC0", which computes the robust sandwich variance treating weights (if supplied) as fixed; "BS", which uses the traditional bootstrap (including re-estimation of the weights, if supplied); "FWB", which uses the fractional weighted bootstrap as implemented in fwb::fwb() (including re-estimation of the weights, if supplied); and "none" to omit calculation of a variance matrix. If NULL (the default), will use "asympt" if weightit is supplied and M-estimation is available and "HC0" otherwise. See the vcov_type component of the outcome object to see which was used.

cluster	optional; for computing a cluster-robust variance matrix, a variable indicating the clustering of observations, a list (or data frame) thereof, or a one-sided formula specifying which variable(s) from the fitted model should be used. Note the cluster-robust variance matrix uses a correction for small samples, as is done in sandwich::vcovCL() by default. Cluster-robust variance calculations are available only when vcov is "asympt", "HC0", "BS", or "FWB".
R	the number of bootstrap replications when vcov is "BS" or "FWB". Default is 500. Ignored otherwise.
offset	optional; a numeric vector containing the model offset. See offset(). An offset can also be present in the model formula.
start	optional starting values for the coefficients.
etastart, mustart	optional starting values for the linear predictor and vector of means. Passed to glm().
control	a list of parameters for controlling the fitting process.
x, y	logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value.
contrasts	an optional list defining contrasts for factor variables. See model.matrix().
fwb.args	an optional list of further arguments to supply to fwb::fwb() when vcov = "FWB".
br	logical; whether to use bias-reduced regression as implemented by brglm2::brglmFit() (including Firth logistic regression). If TRUE, arguments passed to control or ... will be passed to brglm2::brglmControl().
...	arguments to be used to form the default control argument if it is not supplied directly.

Details

glm_weightit() is essentially a wrapper for glm() that optionally computes a coefficient variance matrix that can be adjusted to account for estimation of the weights if a weightit or weightitMSM object is supplied to the weightit argument.

When no argument is supplied to weightit or there is no "Mparts" attribute in the supplied object, the default variance matrix returned will be the "HC0" sandwich variance matrix, which is robust to misspecification of the outcome family (including heteroscedasticity). Otherwise, the default variance matrix uses M-estimation to additionally adjust for estimation of the weights. When possible, this often yields smaller (and more accurate) standard errors. See the individual methods pages to see whether and when an "Mparts" attribute is included in the supplied object. To request that a variance matrix be computed that doesn't account for estimation of the weights even when a compatible weightit object is supplied, set vcov = "HC0", which treats the weights as fixed.

Bootstrapping can also be used to compute the coefficient variance matrix; when vcov = "BS" or vcov = "FWB", which implement the traditional resampling-based and fractional weighted bootstrap, respectively, the entire process of estimating the weights and fitting the outcome model is repeated in bootstrap samples (if a weightit object is supplied). This accounts for estimation of the weights and can be used with any weighting method. It is important to set a seed using set.seed() to ensure replicability of the results. The fractional weighted bootstrap is more reliable but requires

the weighting method to accept sampling weights (which most do, and you'll get an error if it doesn't). Setting `vcov = "FWB"` and supplying `fwb.args = list(wtype = "multinom")` also performs the resampling-based bootstrap but with the additional features **fwb** provides (e.g., a progress bar and parallelization).

Functions in the **sandwich** package can be used to compute standard errors after fitting, regardless of how `vcov` was specified, though these will ignore estimation of the weights, if any. When no adjustment is done for estimation of the weights (i.e., because no `weightit` argument was supplied or there was no `"Mparts"` component in the supplied object), the default variance matrix produced by `glm_weightit()` should align with that from `sandwich::vcovHC(. type = "HC0")` or `sandwich::vcovCL(., type = "HC0", cluster = cluster)` when `cluster` is supplied. Not all types are available for all models.

Value

A `glm_weightit` object, which inherits from `glm`.

Unless `vcov = "none"`, the `vcov` component contains the covariance matrix adjusted for the estimation of the weights if requested and a compatible `weightit` object was supplied. The `vcov_type` component contains the type of variance matrix requested. If `cluster` is supplied, it will be stored in the `"cluster"` attribute of the output object, even if not used.

The `model` component of the output object (also the `model.frame()` output) will include two extra columns when `weightit` is supplied: `(weights)` containing the weights used in the model (the product of the estimated weights and the sampling weights, if any) and `(s.weights)` containing the sampling weights, which will all be 1 if `s.weights` is not supplied in the original `weightit()` call.

See Also

- `lm()` and `glm()` for fitting (generalized) linear models without adjusting standard errors for estimation of the weights.
- `multinom_weightit()` for fitting multinomial regression models that adjust for estimation of the weights.
- `ordinal_weightit()` for fitting ordinal regression models that adjust for estimation of the weights.
- `coxph_weightit()` for fitting Cox proportional hazards models that adjust for estimation of the weights.

Examples

```
data("lalonde", package = "cobalt")

# Logistic regression ATT weights
w.out <- weightit(treat ~ age + educ + married + re74,
                  data = lalonde, method = "glm",
                  estimand = "ATT")

# Linear regression outcome model that adjusts
# for estimation of weights
fit1 <- lm_weightit(re78 ~ treat, data = lalonde,
```

```
                    weightit = w.out)

summary(fit1)

# Linear regression outcome model that treats weights
# as fixed
fit2 <- lm_weightit(re78 ~ treat, data = lalonde,
                    weightit = w.out,
                    vcov = "HC0")

summary(fit2)

# Can also just call summary() with `vcov` option
summary(fit1, vcov = "HC0")

# Linear regression outcome model that bootstraps
# estimation of weights and outcome model fitting
# using fractional weighted bootstrap with "Mammen"
# weights
set.seed(123)
fit3 <- lm_weightit(re78 ~ treat, data = lalonde,
                    weightit = w.out,
                    vcov = "FWB",
                    R = 50, #should use way more
                    fwb.args = list(wtype = "mammen"))

summary(fit3)
```

`glm_weightit-methods` *Methods for* `glm_weightit()` *objects*

Description

This page documents methods for objects returned by `glm_weightit()`, `lm_weightit()`, `ordinal_weightit()`, `multinom_weightit()`, and `coxph_weightit()`. `predict()` methods are described at `predict.glm_weightit()` and `anova()` methods are described at `anova.glm_weightit()`.

Usage

```
## S3 method for class 'glm_weightit'
summary(object, ci = FALSE, level = 0.95, transform = NULL, vcov = NULL, ...)

## S3 method for class 'multinom_weightit'
summary(object, ci = FALSE, level = 0.95, transform = NULL, vcov = NULL, ...)

## S3 method for class 'ordinal_weightit'
summary(
  object,
```

```
  ci = FALSE,
  level = 0.95,
  transform = NULL,
  thresholds = TRUE,
  vcov = NULL,
  ...
)

## S3 method for class 'coxph_weightit'
summary(object, ci = FALSE, level = 0.95, transform = NULL, vcov = NULL, ...)

## S3 method for class 'glm_weightit'
print(x, digits = max(3L, getOption("digits") - 3L), ...)

## S3 method for class 'glm_weightit'
vcov(object, complete = TRUE, vcov = NULL, ...)

## S3 method for class 'glm_weightit'
estfun(x, asympt = TRUE, ...)

## S3 method for class 'glm_weightit'
update(object, formula. = NULL, ..., evaluate = TRUE)
```

Arguments

`object`, `x`	an output from one of the above modeling functions.
`ci`	`logical`; whether to display Wald confidence intervals for estimated coefficients. Default is `FALSE`. (Note: this argument can also be supplied as `conf.int`.)
`level`	when `ci = TRUE`, the desired confidence level.
`transform`	the function used to transform the coefficients, e.g., `exp` (which can also be supplied as a string, e.g., `"exp"`); passed to `match.fun()` before being used on the coefficients. When `ci = TRUE`, this is also applied to the confidence interval bounds. If specified, the standard error will be omitted from the output. Default is no transformation.
`vcov`	either a string indicating the method used to compute the variance of the estimated parameters for `object`, a function used to extract the variance, or the variance matrix itself. Default is to use the variance matrix already present in `object`. If a string or function, arguments passed to `...` are supplied to the method or function. (Note: for `vcov()`, can also be supplied as `type`.)
`...`	for `vcov()` or `summary()` or `confint()` with `vcov` supplied, other arguments used to compute the variance matrix depending on the method supplied to `vcov`, e.g., `cluster`, `R`, or `fwb.args`. For `update()`, additional arguments to the call or arguments with changed values. See `glm_weightit()` for details.
`thresholds`	`logical`; whether to include thresholds in the `summary()` output for `ordinal_weightit` objects. Default is `TRUE`.
`digits`	the number of *significant* digits to be passed to `format(coef(x), .)` when `print()`ing.

complete	logical; whether the full variance-covariance matrix should be returned also in case of an over-determined system where some coefficients are undefined and coef(.) contains NAs correspondingly. When complete = TRUE, vcov() is compatible with coef() also in this singular case.
asympt	logical; for estfun(), whether to use the asymptotic empirical estimating functions that account for estimation of the weights (when Mparts is available). Default is TRUE. Set to FALSE to ignore estimation of the weights. Ignored when Mparts is not available or no argument was supplied to weightit in the fitting function.
formula.	changes to the model formula, passed to the new argument of update.formula().
evaluate	logical; whether to evaluate the call (TRUE, the default) or just return it.

Details

vcov() by default extracts the parameter covariance matrix already computed by the fitting function, and summary() and confint() uses this covariance matrix to compute standard errors and Wald confidence intervals (internally calling confint.lm()), respectively. Supplying arguments to vcov or ... will compute a new covariance matrix. If cluster was supplied to the original fitting function, it will be incorporated into any newly computed covariance matrix unless cluster = NULL is specified in vcov(), summary(), or confint(). For other arguments (e.g., R and fwb.args), the defaults are those used by glm_weightit(). Note that for vcov = "BS" and vcov = "FWB" (and vcov = "const" for multinom_weightit or ordinal_weightit objects), the environment for the fitting function is used, so any changes to that environment may affect calculation. It is always safer to simply recompute the fitted object with a new covariance matrix than to modify it with the vcov argument, but it can be quicker to just request a new covariance matrix when refitting the model is slow.

update() updates a fitted model object with new arguments, e.g., a new model formula, dataset, or variance matrix. When only arguments that control the computation of the variance are supplied, only the variance will be recalculated (i.e., the parameters will not be re-estimated). When data is supplied, weightit is not supplied, and a weightit object was originally passed to the model fitting function, the weightit object will be re-fit with the new dataset before the model is refit using the new weights and new data. That is, calling update(obj, data = d) is equivalent to calling update(obj, data = d, weightit = update(obj$weightit, data = d)) when a weightit object was supplied to the model fitting function. Similarly, supplying s.weights or weights passes the argument through to weightit() to be refit. When s.weights or weights are supplied and no weightit object is present, a fake one containing just the supplied weights will be created.

estfun() extracts the empirical estimating functions for the fitted model, optionally accounting for the estimation of the weights (if available). This, along with bread(), is used by sandwich::sandwich() to compute the robust covariance matrix of the estimated coefficients. See glm_weightit() and vcov() above for more details.

Value

summary() returns a summary.glm_weightit() object, which has its own print() method. For coxph_weightit() objects, the print() and summary() methods are more like those for glm objects than for coxph objects.

Otherwise, all methods return the same type of object as their generics.

See Also

glm_weightit() for the page documenting glm_weightit(), lm_weightit(), ordinal_weightit(), multinom_weightit(), and coxph_weightit(). summary.glm(), vcov(), confint() for the relevant methods pages. predict.glm_weightit() for computing predictions from the models. anova.glm_weightit() for comparing models using a Wald test.

sandwich::estfun() and sandwich::bread() for the estfun() and bread() generics.

Examples

```
## See examples at ?glm_weightit
```

---

make_full_rank *Make a design matrix full rank*

Description

When writing user-defined methods for use with weightit(), it may be necessary to take the potentially non-full rank covs data frame and make it full rank for use in a downstream function. This function performs that operation.

Usage

```
make_full_rank(mat, with.intercept = TRUE)
```

Arguments

mat	a numeric matrix or data frame to be transformed. Typically this contains covariates. NAs are not allowed.
with.intercept	logical; whether an intercept (i.e., a vector of 1s) should be added to mat before making it full rank. If TRUE, the intercept will be used in determining whether a column is linearly dependent on others. Regardless, no intercept will be included in the output.

Details

make_full_rank() calls qr() to find the rank and linearly independent columns of mat, which are retained while others are dropped. If with.intercept is set to TRUE, an intercept column is added to the matrix before calling qr(). Note that dependent columns that appear later in mat will be dropped first.

See example at method_user.

Value

An object of the same type as mat containing only linearly independent columns.

Note

Older versions would drop all columns that only had one value. With `with.intercept = FALSE`, if only one column has only one value, it will not be removed, and it will function as though there was an intercept present; if more than one column has only one value, only the first one will remain.

See Also

`method_user`, `model.matrix()`

Examples

```
set.seed(1234)
n <- 20
c1 <- rbinom(n, 1, .4)
c2 <- 1 - c1
c3 <- rnorm(n)
c4 <- 10 * c3
mat <- data.frame(c1, c2, c3, c4)

make_full_rank(mat) #keeps c1 and c3

make_full_rank(mat, with.intercept = FALSE) #keeps c1, c2, and c3
```

---

method_bart *Propensity Score Weighting Using BART*

Description

This page explains the details of estimating weights from Bayesian additive regression trees (BART)-based propensity scores by setting `method = "bart"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating propensity scores using BART and then converting those propensity scores into weights using a formula that depends on the desired estimand. This method relies on `dbarts::bart2()` from the **dbarts** package.

Binary Treatments:

For binary treatments, this method estimates the propensity scores using `dbarts::bart2()`. The following estimands are allowed: ATE, ATT, ATC, ATO, ATM, and ATOS. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Multi-Category Treatments:

For multi-category treatments, the propensity scores are estimated using several calls to `dbarts::bart2()`, one for each treatment group; the treatment probabilities are not normalized to sum to 1. The following estimands are allowed: ATE, ATT, ATC, ATO, and ATM. The weights for each estimand are computed using the standard formulas. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Continuous Treatments:

For continuous treatments, weights are estimated as $w_i = f_A(a_i)/f_{A|X}(a_i)$, where $f_A(a_i)$ (known as the stabilization factor) is the unconditional density of treatment evaluated the observed treatment value and $f_{A|X}(a_i)$ (known as the generalized propensity score) is the conditional density of treatment given the covariates evaluated at the observed treatment value. The shape of $f_{A|X}(.)$ is controlled by the `density` argument described below (normal distribution by default), and the predicted values used for the mean of the conditional density are estimated using BART as implemented in `dbarts::bart2()`. $f_A(.)$ is estimated by marginalizing over $f_{A|X}(.)$. Kernel density estimation can be used instead of assuming a specific density for the denominator by setting `density = "kernel"`. Other arguments to `density()` can be specified to refine the density estimation parameters.

Multilevel Treatment Models:

When the model `formula` contains **lme4**-style random effects terms (e.g., `treat ~ x1 + x2 + (1 | school)`), a multilevel BART model is fit using `stan4bart::stan4bart()` instead of `dbarts::bart2()`. This combines a BART sum-of-trees for the covariates with a Stan-estimated random-effects component and can improve balance when units are clustered (e.g., patients within hospitals or students within schools). The full flexibility of `lme4::glmer()` random effects is available, including multiple grouping factors (e.g., `(1 | school) + (1 | district)`) and random slopes. The grouping (and any random-slope) variables are taken from `data`; the covariates are placed in the BART component internally (i.e., the fitted model is `treat ~ bart(x1 + x2) + (1 | school)`). The estimated propensity scores (or conditional means for continuous treatments) include the estimated random effects; i.e., they are cluster-specific.

`stan4bart()` uses a different set of control arguments from `bart2()`: the number of posterior draws and warmup iterations are controlled by `iter` and `warmup`, the number of chains and parallel workers by `chains` and `cores`, BART hyperparameters (e.g., `n.trees`) are supplied in a `bart_args` list, and Stan and prior options in a `stan_args` list; see `stan4bart::stan4bart()`. For convenience, `bart2()`-style arguments are translated to their `stan4bart()` equivalents when supplied (e.g., `n.chains` to `chains`, `n.threads` to `cores`, and BART hyperparameters like `n.trees` to `bart_args`), so the same argument names can be used with or without random effects. Only the arguments described here are passed on; anything else, including the `bart2()` arguments listed as exceptions in *Additional Arguments* below, is ignored. As with the single-level case, M-estimation is not supported.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, a single BART model of the probability of being censored is fit, and the weights are `1/P(C = 0 | X)` for the units still under observation and 0 for the censored units. The returned propensity score is the probability of *being censored*.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point.

Sampling Weights:

Sampling weights are not supported.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians. The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Details

BART works by fitting a sum-of-trees model for the treatment or probability of treatment. The number of trees is determined by the `n.trees` argument. Bayesian priors are used for the hyperparameters, so the result is a posterior distribution of predicted values for each unit. The mean of these for each unit is taken for use in computing the (generalized) propensity score. Although the hyperparameters governing the priors can be modified by supplying arguments to `weightit()` that are passed to the BART fitting function, the default values tend to work well and require little modification (though the defaults differ for continuous and categorical treatments; see the `dbarts::bart2()` documentation for details). Unlike many other machine learning methods, no loss function is optimized and the hyperparameters do not need to be tuned (e.g., using cross-validation), though performance can benefit from tuning. BART tends to balance sparseness with flexibility by using very weak learners as the trees, which makes it suitable for capturing complex functions without specifying a particular functional form and without overfitting.

Reproducibility:

BART has a random component, so some work must be done to ensure reproducibility across runs. See the *Reproducibility* section at `dbarts::bart2()` for more details. To ensure reproducibility, one can do one of two things:

1. supply an argument to `seed`, which is passed to `dbarts::bart2()` and sets the seed for single- and multi-threaded uses, or
2. call `set.seed()` and set `n.threads = 1` to use single-threading.

Note that to ensure reproducibility on any machine, regardless of the number of cores available, one should use single-threading by setting `n.threads = 1` and either supply `seed` or call `set.seed()`.

Additional Arguments

All arguments to `dbarts::bart2()` can be passed through `weightit()` or `weightitMSM()`, with the following exceptions:

- `test`, `weights`, `subset`, `offset.test`, and `samplerOnly` are ignored
- `combineChains` is always set to `TRUE`
- `keepCall` is always set to `FALSE`
- `verbose` is always set to `FALSE` (use `weightit()`'s own `verbose` argument instead)

When the model `formula` contains random effects terms, `stan4bart::stan4bart()` is used instead of `bart2()`, and the arguments it accepts differ; see *Multilevel Treatment Models* below.

See also the *Reproducibility* section below for information on using the `seed` argument.

The following additional argument may be supplied:

`use.offset` `logical`; whether to use the linear predictor resulting from a generalized linear model as an offset to the BART model. If `TRUE`, this fits a probit regression model (for binary treatments) or a linear regression model (for continuous treatments) and supplies the linear predictor to the `offset` argument of `bart2()` or `stan4bart()`, overriding any supplied argument to `offset`. Default is `FALSE` to omit this offset. Only allowed for binary and continuous treatments.

For binary and multi-category treatments, the following arguments may be supplied:

`subclass` `integer`; the number of subclasses to use for computing weights using marginal mean weighting through stratification (MMWS). If `NULL`, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and weights will be computed based on subclass membership. See `get_w_from_ps()` for details and references.

For continuous treatments, the following arguments may be supplied:

`density` A function corresponding to the conditional density of the treatment. The standardized residuals of the treatment model will be fed through this function to produce the denominator of the generalized propensity score weights. If blank, `dnorm()` is used as recommended by Robins et al. (2000). This can also be supplied as a string containing the name of the function to be called. If the string contains underscores, the call will be split by the underscores and the latter splits will be supplied as arguments to the second argument and beyond. For example, if `density = "dt_2"` is specified, the density used will be that of a t-distribution with 2 degrees of freedom. Using a t-distribution can be useful when extreme outcome values are observed (Naimi et al., 2014).

Can also be `"kernel"` to use kernel density estimation, which calls `density()` to estimate the denominator density for the weights. (This used to be requested by setting `use.kernel = TRUE`, which is now deprecated.)

`bw`, `adjust`, `kernel`, `n` If `density = "kernel"`, the arguments to `density()`. The defaults are the same as those in `density()`.

Additional Outputs

`obj` When `include.obj = TRUE`, the `bart2` fit(s) used to generate the predicted values (or the `stan4bart::stan4bart()` fit(s) when the model `formula` contains random effects terms). With multi-category treatments, this will be a list of the fits; otherwise, it will be a single fit. The predicted probabilities used to compute the propensity scores can be extracted using `fitted()`.

References

Hill, J., Weiss, C., & Zhai, F. (2011). Challenges With Propensity Score Strategies in a High-Dimensional Setting and a Potential Alternative. *Multivariate Behavioral Research*, 46(3), 477–513. doi:10.1080/00273171.2011.570161

Chipman, H. A., George, E. I., & McCulloch, R. E. (2010). BART: Bayesian additive regression trees. *The Annals of Applied Statistics*, 4(1), 266–298. doi:10.1214/09AOAS285

Note that many references that deal with BART for causal inference focus on estimating potential outcomes with BART, not the propensity scores, and so are not directly relevant when using BART to estimate propensity scores for weights.

See method_glm for additional references on propensity score weighting more generally.

See Also

weightit(), weightitMSM(), get_w_from_ps()

method_super for stacking predictions from several machine learning methods, including BART.

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "bart", estimand = "ATT"))

summary(W1)

cobalt::bal.tab(W1)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                nodegree + re74, data = lalonde,
                method = "bart", estimand = "ATE"))

summary(W2)

cobalt::bal.tab(W2)

#Balancing covariates with respect to re75 (continuous)
#with kernel density estimation for GPS
(W3 <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "bart", density = "kernel"))

summary(W3)

cobalt::bal.tab(W3)
```

Description

This page explains the details of estimating weights from covariate balancing propensity scores by setting `method = "cbps"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating propensity scores using generalized method of moments and then converting those propensity scores into weights using a formula that depends on the desired estimand. This method relies on code written for **WeightIt** using `optim()`.

Binary Treatments:

For binary treatments, this method estimates the propensity scores and weights using `optim()` using formulas described by Imai and Ratkovic (2014). The following estimands are allowed: ATE, ATT, ATC, and ATO.

Multi-Category Treatments:

For multi-category treatments, this method estimates the generalized propensity scores and weights using `optim()` using formulas described by Imai and Ratkovic (2014). The following estimands are allowed: ATE, ATT, and ATO.

Continuous Treatments:

For continuous treatments, this method estimates the generalized propensity scores and weights using `optim()` using a modification of the formulas described by Fong, Hazlett, and Imai (2018). See Details.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, the balance conditions require the weighted covariate means of the units still under observation to equal those of the full at-risk sample. The censored units receive a weight of 0.

Censoring weights are also available with `is.MSM.method = TRUE`, i.e., when the weights at all time points are estimated simultaneously; `"cbps"` is the only method for which this is possible. Each time point's balance condition is then evaluated among the units still under observation at that time point, so the covariates and treatments that are missing for already-censored units never enter. The treatment conditions use the cumulative weights, which include the censoring factors; each censoring condition uses its own factor. As for treatments, M-estimation is not available with `is.MSM.method = TRUE`.

Longitudinal Treatments:

For longitudinal treatments, the weights are computed using methods similar to those described by Huffman and van Gameren (2018). This involves specifying moment conditions for the models at each time point as with single-time point treatments but using the product of the time-specific weights as the weights that are used in the balance moment conditions. This yields weights that balance the covariates at each time point. This is not the same implementation as is implemented in `CBPS::CBMSM()`, and results should not be expected to align between the two methods. Any combination of treatment types is supported.

For the over-identified version (i.e., setting `over = TRUE`), the empirical variance is used in the objective function, whereas the expected variance averaging over the treatment is used with binary and multi-category point treatments.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is supported for the just-identified CBPS (the default, setting `over = FALSE`) for binary and multi-category treatments and censoring weights. Otherwise (i.e., for continuous or longitudinal treatments or when `over = TRUE`), M-estimation is not supported. See `glm_weightit()` and `vignette("estimating-effects")` for details.

Details

CBPS estimates the coefficients of a generalized linear model (for binary treatments and censoring weights), multinomial logistic regression model (for multi-category treatments), or linear regression model (for continuous treatments) that is used to compute (generalized) propensity scores, from which the weights are computed. It involves replacing (or augmenting, in the case of the over-identified version) the standard maximum likelihood score equations with the balance constraints in a generalized method of moments estimation. The idea is to nudge the estimation of the coefficients toward those that produce balance in the weighted sample. The just-identified version (with `over = FALSE`) does away with the maximum likelihood score equations for the coefficients so that only the balance constraints are used, which will therefore produce superior balance on the means (i.e., corresponding to the balance constraints) for binary and multi-category treatments and linear terms for continuous treatments than will the over-identified version.

Just-identified CBPS is very similar to entropy balancing and inverse probability tilting. For the ATT, all three methods will yield identical estimates. For other estimands, the results will differ.

Note that **WeightIt** provides different functionality from the **CBPS** package in terms of the versions of CBPS available; for extensions to CBPS (e.g., optimal CBPS and CBPS for instrumental variables), the **CBPS** package may be preferred. Note that for longitudinal treatments, `CBPS::CBMSM()` uses different methods and produces different results from `weightitMSM()` called with `method = "cbps"`.

Additional Arguments

The following additional arguments can be specified:

`over` `logical`; whether to request the over-identified CBPS, which combines the generalized linear model regression score equations (for binary treatments and censoring weights), multinomial logistic regression score equations (for multi-category treatments), or linear regression score equations (for continuous treatments) to the balance moment conditions. Default is `FALSE` to use the just-identified CBPS.

`twostep` `logical`; when `over = TRUE`, whether to use the two-step approximation to the generalized method of moments variance. Default is `TRUE`. Setting to `FALSE` increases computation time but may improve estimation. Ignored with a warning when `over = FALSE`.

`link` the link used in the generalized linear model for the propensity scores when treatment is binary. Default is `"logit"` for logistic regression, which is used in the original description of the method by Imai and Ratkovic (2014), but others are allowed, including `"probit"`, `"cauchit"`, `"cloglog"`, `"loglog"`, `"log"`, `"clog"`, `"softplus"`, and `"identity"`. Note that negative weights are possible with these last three and they should be used with caution. An object of class `"link-glm"` can also be supplied. The argument is passed to `quasibinomial()`. Ignored for multi-category, continuous, and longitudinal treatments.

`reltol` the relative tolerance for convergence of the optimization. Passed to the `control` argument of `optim()`. Default is `1e-10`.

`maxit` the maximum number of iterations for convergence of the optimization. Passed to the `control` argument of `optim()`. Default is 5000 for binary treatments and censoring, 1000 for multi-category treatments, and 10000 for continuous and longitudinal treatments.

`solver` the solver to use to estimate the parameters of the just-identified CBPS. Allowable options include `"multiroot"` to use `rootSolve::multiroot()` and `"optim"` to use `stats::optim()`. `"multiroot"` is the default when **rootSolve** is installed, as it tends to be much faster and more accurate; otherwise, `"optim"` is the default and requires no dependencies. Regardless of `solver`, the output of `optim()` is returned when `include.obj = TRUE` (see below). When `over = TRUE`, the parameter estimates of the just-identified CBPS are used as starting values for the over-identified CBPS.

`moments` `integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 1 to balance covariate means.

`int` `logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates. Only allowed with binary and multi-category treatments.

Additional Outputs

`obj` When `include.obj = TRUE`, the output of the final call to `optim()` used to produce the model parameters. Note that because of variable transformations, the resulting parameter estimates may not be interpretable.

Note

This method used to rely on functionality in the **CBPS** package, but no longer does. Slight differences may be found between the two packages in some cases due to numerical imprecision (or, for continuous and longitudinal treatments, due to a difference in the estimator). **WeightIt** supports arbitrary numbers of groups for the multi-category CBPS and any estimand, whereas **CBPS** only supports up to four groups and only the ATE. The implementation of the just-identified CBPS for continuous treatments also differs from that of **CBPS**, and departs slightly from that described by Fong et al. (2018). The treatment mean and treatment variance are treated as random parameters

to be estimated and are included in the balance moment conditions. In Fong et al. (2018), the treatment mean and variance are fixed to their empirical counterparts. For continuous treatments with the over-identified CBPS, **WeightIt** and **CBPS** use different methods of specifying the GMM variance matrix, which may lead to differing results.

Note that the default method differs between the two implementations; by default **WeightIt** uses the just-identified CBPS, which is faster to fit, yields better balance, and is compatible with M-estimation for estimating the standard error of the treatment effect, whereas **CBPS** uses the over-identified CBPS by default. However, both the just-identified and over-identified versions are available in both packages.

When the **rootSolve** package is installed, the optimization process will be slightly faster and more accurate because starting values are provided by an initial call to `rootSolve::multiroot()`. However, the package is not required.

Unsolvable Balance Conditions:

The just-identified CBPS (`over = FALSE`) has exactly as many parameters as balance conditions, so at a solution the conditions hold exactly and the corresponding covariate moments are exactly balanced. For some samples no such solution exists: the optimizer settles at a point where the conditions are still unmet, and the resulting weights balance the covariates only approximately. This is most likely with a continuous treatment and many covariates, and whether it happens depends on the sample, not just on the specification.

`weightit()` warns when this occurs, since the returned weights are otherwise indistinguishable from ones that did solve the conditions. When it does, assess the balance actually achieved (e.g., with `cobalt::bal.tab()`) rather than assuming it is exact. With `include.obj = TRUE`, the `value` component of the returned `optim()` output object is the norm of the mean estimating function, which is what the warning is based on: it is essentially 0 when the conditions were solved.

References

Binary treatments:

Imai, K., & Ratkovic, M. (2014). Covariate balancing propensity score. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 76(1), 243–263.

Multi-Category treatments:

Imai, K., & Ratkovic, M. (2014). Covariate balancing propensity score. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 76(1), 243–263.

Continuous treatments:

Fong, C., Hazlett, C., & Imai, K. (2018). Covariate balancing propensity score for a continuous treatment: Application to the efficacy of political advertisements. *The Annals of Applied Statistics*, 12(1), 156–177. doi:10.1214/17AOAS1101

Longitudinal treatments:

Huffman, C., & van Gameren, E. (2018). Covariate Balancing Inverse Probability Weights for Time-Varying Continuous Interventions. *Journal of Causal Inference*, 6(2). doi:10.1515/jci2017-0002

Note: one should not cite Imai & Ratkovic (2015) when using CBPS for longitudinal treatments.

Some of the code was inspired by the source code of the **CBPS** package.

See Also

weightit(), weightitMSM()

method_ebal and method_ipt for entropy balancing and inverse probability tilting, which work similarly.

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1a <- weightit(treat ~ age + educ + married +
                   nodegree + re74, data = lalonde,
                 method = "cbps", estimand = "ATT"))

summary(W1a)

cobalt::bal.tab(W1a)

#Balancing covariates between treatment groups (binary)
#using over-identified CBPS with probit link
(W1b <- weightit(treat ~ age + educ + married +
                   nodegree + re74, data = lalonde,
                 method = "cbps", estimand = "ATT",
                 over = TRUE, link = "probit"))

summary(W1b)

cobalt::bal.tab(W1b)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                   nodegree + re74, data = lalonde,
                 method = "cbps", estimand = "ATE"))

summary(W2)

cobalt::bal.tab(W2)

#Balancing covariates with respect to re75 (continuous)
(W3 <- weightit(re75 ~ age + educ + married +
                   nodegree + re74, data = lalonde,
                 method = "cbps"))

summary(W3)

cobalt::bal.tab(W3)

#Longitudinal treatments
data("msmdata")
(W4 <- weightitMSM(list(A_1 ~ X1_0 + X2_0,
                        A_2 ~ X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0),
```

```
                    data = msmdata,
                    method = "cbps"))

summary(W4)

cobalt::bal.tab(W4)
```

`method_cfd` *Characteristic Function Distance Balancing*

Description

This page explains the details of estimating weights using characteristic function distance (CFD) balancing by setting `method = "cfd"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary and multi-category treatments, as well as for estimating censoring weights.

In general, this method relies on estimating weights by minimizing a scalar measure of covariate balance, the CFD. The CFD is related to the maximum mean discrepancy and captures covariate balance for the joint covariate distribution as determined by a specific choice of kernel. This method relies on code written for **WeightIt** using `osqp::osqp()` from the **osqp** package to perform the optimization. This method may be slow or memory-intensive for large datasets.

Binary Treatments:

For binary treatments, this method estimates the weights using `osqp()` using formulas described by Santra, Chen, and Park (2026). The following estimands are allowed: ATE, ATT, and ATC.

Multi-Category Treatments:

For multi-category treatments, this method estimates the weights using `osqp()` using formulas described by Santra, Chen, and Park (2026). The following estimands are allowed: ATE and ATT.

Continuous Treatments:

CFD balancing is not compatible with continuous treatments.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, the CFD minimized is that between the weighted units still under observation and the full at-risk sample (censored and uncensored combined), rather than that between two treatment groups. Weights are estimated for the units still under observation only; the censored units receive a weight of 0.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point. This method is not guaranteed to yield optimal balance at each time point. **NOTE: the use of CFD balancing with longitudinal treatments has not been validated!** Because of this, `weightitMSM()` errors when this method is requested; set `weightit.force = TRUE` to bypass that error.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios. In some cases, sampling weights will cause the optimization to fail due to lack of convexity or infeasible constraints.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Details

CFD balancing is a method of estimating weights using optimization without a propensity score. The weights are the solution to a constrained quadratic optimization problem where the objective function concerns covariate balance as measured by the CFD between groups.

CFD balancing for binary and multi-category treatments involves minimizing the CFD between the treatment groups and between each treatment group and a target group (e.g., the full sample for the ATE). The CFD is a scalar measure of the difference between two multivariate distributions. The performance of CFD balance depends on the choice of kernel, controlled by the `kernel` argument. Each kernel corresponds to different assumptions about the form of the true outcome model. See Santra et al. (2026) for a comparison of these different kernels. Setting `kernel = "energy"` is equivalent to energy balancing, which can also be requested by using `method = "energy"`.

The primary benefit of CFD balancing is that all features of the covariate distribution are balanced, not just means, unlike with other optimization-based methods like entropy balancing. Still, it is possible to add additional balance constraints to require balance on individual terms using the `moments` argument, as with entropy balancing. CFD balancing can sometimes yield weights with high variability; the `lambda` argument can be supplied to penalize highly variable weights to increase the effective sample size at the expense of balance.

Reproducibility:

Although there are no stochastic components to the optimization, **osqp** by default updates the optimization based on how long the optimization has been running, which will vary across runs even when a seed is set and no parameters have been changed. See the discussion here for more details. To ensure reproducibility by default, `adaptive_rho_interval` is set to 50, which fixes the update schedule at a set number of iterations. See `osqp::osqpSettings()` for details.

Additional Arguments

The following additional arguments can be specified:

`kernel` the name of the kernel used to characterize the CFD. Allowable options include `"gaussian"` for the multivariate Gaussian kernel (the default), `"matern"` for the multivariate Matern kernel, `"energy"` for the energy distance kernel, `"laplace"` for the univariate Laplacian kernel, and `"t"` for the univariate t-distribution kernel.

`nu` for `kernel = "matern"`, the ν parameter used to control smoothness. The default value is 3/2. For any values other than 1/2, 3/2, and 5/2, the **GPBayes** package is required to compute the Matern kernel. For `kernel = "t"`, the degrees of freedom for the univariate t-distributions used in the kernel. The default value is 5. Ignored for other kernels.

`nsim` for `kernel = "t"`, the number of simulations to use to compute the t-distribution kernel. Default is 5000. Greater is better but takes longer and uses more memory.

`lambda` a positive numeric scalar used to penalize the square of the weights. This value divided by the square of the total sample size is added to the diagonal of the quadratic part of the loss function. Higher values favor weights with less variability. Default is .0001, which is essentially 0.

`moments` `integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 0 to impose no constraint on balance.

`int` `logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`tols` when `moments` is positive, a number corresponding to the maximum allowed standardized mean difference allowed. Default is 0. Ignored when `moments = 0`.

`min.w` the minimum allowable weight. Negative values (including `-Inf`) are allowed. Default is `1e-8`.

`bw_scale` a positive numeric scalar multiplying the kernel bandwidth, which is set by default to the median of the pairwise distances between units. Values below 1 make the kernel more local and values above 1 make it more diffuse. Default is 1. Ignored when `kernel = "energy"`, which has no bandwidth.

For binary and multi-category treatments, the following additional arguments can be specified:

`improved` `logical`; whether to include an additional term in the CFD objective function to minimize the distance between pairs of groups when `estimand = "ATE"`. Default is `TRUE`.

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates.

The `moments` argument functions differently for `method = "cfd"` from how it does with some other methods. When unspecified or set to zero, CFD balancing weights are estimated as described by Santra et al. (2026) for binary and multi-category treatments. When `moments` is set to an integer larger than 0, additional balance constraints on the requested moments of the covariates are also included, guaranteeing exact moment balance on these covariates while minimizing the CFD of the weighted sample. This involves exact balance on the means of the entered covariates. The constraint on exact balance can be relaxed using the `tols` argument.

Any other arguments will be passed to `osqp::osqpSettings()`. Some defaults differ from those in `osqpSettings()`; see *Reproducibility* section.

Additional Outputs

`obj` When `include.obj = TRUE`, the output of the call to `osqp::solve_osqp()`, which contains the dual variables and convergence information.

Note

Sometimes the optimization can fail to converge because the problem is not convex. A warning will be displayed if so. In these cases, try simply re-fitting the weights without changing anything (but see the *Reproducibility* section above). If the method repeatedly fails, you should try another method or change the supplied parameters (though this is uncommon). Increasing `max_iter` or changing `adaptive_rho_interval` might help.

If it seems like the weights are balancing the covariates but you still get a failure to converge, this usually indicates that more iterations are needed to find the optimal solution. This can occur when `moments` or `int` are specified. `max_iter` should be increased, and setting `verbose = TRUE` allows you to monitor the process and examine if the optimization is approaching convergence.

If `min.w` is positive and you still get a warning about the presence of negative weights, try setting `eps` to a smaller number (e.g., to `1e-12`).

As of version 1.5.0, `polish` is now set to `TRUE` by default. This should yield slightly improved solutions but may be a little slower.

References

Santra, D., Chen, G., & Park, C. (2026). Distributional Balancing for Causal Inference: A Unified Framework via Characteristic Function Distance (arXiv:2601.15449). arXiv. doi:10.48550/arXiv.2601.15449

See Also

`weightit()`, `weightitMSM()`

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "cfd", estimand = "ATE"))

summary(W1)

cobalt::bal.tab(W1)

#Using a different kernel:
(W1b <- weightit(treat ~ age + educ + married +
                   nodegree + re74, data = lalonde,
                 method = "cfd", estimand = "ATE",
                 kernel = "matern", nu = 5/2))
```

```
summary(W1b)

cobalt::bal.tab(W1b)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "cfd", estimand = "ATT",
                focal = "black"))

summary(W2)

cobalt::bal.tab(W2)
```

`method_ebal` *Entropy Balancing*

Description

This page explains the details of estimating weights using entropy balancing by setting `method = "ebal"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating weights by minimizing the negative entropy of the weights subject to exact moment balancing constraints. This method relies on code written for **WeightIt** using `optim()`.

Binary Treatments:

For binary treatments, this method estimates the weights using `optim()` using formulas described by Hainmueller (2012). The following estimands are allowed: ATE, ATT, and ATC. When the ATE is requested, the optimization is run twice, once for each treatment group.

Multi-Category Treatments:

For multi-category treatments, this method estimates the weights using `optim()`. The following estimands are allowed: ATE and ATT. When the ATE is requested, the optimization is run once for each treatment group. When the ATT is requested, the optimization is run once for each non-focal (i.e., control) group.

Continuous Treatments:

For continuous treatments, this method estimates the weights using `optim()` using formulas described by Tübbicke (2022) and Vegetabile et al. (2021).

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, entropy balancing is performed on the units still under observation only, with the target being the covariate means of the full at-risk sample (censored and uncensored combined). Because a single set of weights is estimated rather than one per treatment group, the optimization problem is smaller than for the ATE. The censored units receive a weight of 0. M-estimation is supported when `tols` is 0. With `link = "clog"`, `method = "ipt"` yields identical weights.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point. This method is not guaranteed to yield exact balance at each time point. **NOTE: the use of entropy balancing with longitudinal treatments has not been validated and should not be done!** Because of this, `weightitMSM()` errors when this method is requested; set `weightit.force = TRUE` to bypass that error.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is supported for all treatment types and estimands except when `tols` is greater than 0. See `glm_weightit()` and `vignette("estimating-effects")` for details.

Details

Entropy balancing involves the specification of an optimization problem, the solution to which is then used to compute the weights. The constraints of the primal optimization problem correspond to covariate balance on the means (for binary and multi-category treatments) or treatment-covariate covariances (for continuous treatments), positivity of the weights, and that the weights sum to a certain value. It turns out that the dual optimization problem is much easier to solve because it is over only as many variables as there are balance constraints rather than over the weights for each unit, and it is unconstrained.

Zhao and Percival (2017) found that entropy balancing for the ATT of a binary treatment actually involves the estimation of the coefficients of a logistic regression propensity score model but using a specialized loss function different from that optimized with maximum likelihood. Entropy balancing is doubly robust (for the ATT) in the sense that it is consistent either when the true propensity score model is a logistic regression of the treatment on the covariates or when the true outcome model for the control units is a linear regression of the outcome on the covariates, and it attains a semi-parametric efficiency bound when both are true. Entropy balancing will always yield exact mean balance on the included terms unless an imbalance tolerance is requested with `tols`.

When `tols` is greater than 0, inexact balance is allowed on the covariates. This can improve precision while allowing a small amount of bias. The optimization problem is an L1 regularization problem and is solved using the Fast Iterative Shrinkage-Thresholding Algorithm (FISTA).

Additional Arguments

`base.weights` a vector of base weights, one for each unit. These correspond to the base weights q in Hainmueller (2012). The estimated weights minimize the Kullback entropy divergence from the base weights, defined as $\sum w \log(w/q)$, subject to exact balance constraints. These can be used to supply previously estimated weights so that the newly estimated weights retain some of the properties of the original weights while ensuring the balance constraints are met. Sampling weights should not be passed to `base.weights` but can be included in a `weightit()` call that includes `s.weights`. Can also be supplied as `b.weights` or `base.weight`.

`reltol` the relative tolerance for convergence of the optimization. Passed to the `control` argument of `optim()`. Default is `1e-10`.

`maxit` the maximum number of iterations for convergence of the optimization. Passed to the `control` argument of `optim()`. Default is 10000.

`solver` the solver to use to estimate the parameters. Allowable options include `"multiroot"` to use `rootSolve::multiroot()` and `"optim"` to use `stats::optim()`. `"multiroot"` is the default when **rootSolve** is installed, as it tends to be much faster and more accurate; otherwise, `"optim"` is the default and requires no dependencies. The output of whichever solver was used is returned when `include.obj = TRUE` (see below).

`moments` `integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 1 to balance covariate means.

`int` `logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates. Only allowed with binary and multi-category treatments.

`tols` a number corresponding to the maximum allowed standardized mean difference (for binary and multi-category treatments) or treatment-covariate correlation (for continuous treatments) allowed. Default is 0 for exact balance.

`d.moments` `integer`; with continuous treatments, the number of moments of the treatment and covariate distributions that are constrained to be the same in the weighted sample as in the original sample. For example, setting `d.moments = 3` ensures that the mean, variance, and skew of the treatment and covariates are the same in the weighted sample as in the unweighted sample. `d.moments` acts as a floor: the number of moments held for each covariate is the larger of `d.moments` and that covariate's entry in `moments`, so a value less than or equal to `moments` has no effect on the covariates and a larger one raises them all, including any not named when `moments` is supplied as a per-covariate vector. `d.moments` alone controls how many moments of the treatment are held; the default of 1 holds only its mean. Vegetabile et al. (2021) recommend setting `d.moments = 3`, even if `moments` is less than 3. This argument corresponds to the tuning parameters r and s in Vegetabile et al. (2021) (which here are set to be equal). Ignored for binary and multi-category treatments.

The `stabilize` argument is ignored; in the past it would reduce the variability of the weights through an iterative process. If you want to minimize the variance of the weights subject to balance constraints, use `method = "optweight"`.

Additional Outputs

`obj` When `include.obj = TRUE`, the output of the call to the solver (see `solver` above), which contains the dual variables and convergence information. For ATE fits or with multi-category treatments, a list of such outputs, one for each weighted group.

References

Binary Treatments:

`estimand = "ATT"`*:*

Hainmueller, J. (2012). Entropy Balancing for Causal Effects: A Multivariate Reweighting Method to Produce Balanced Samples in Observational Studies. *Political Analysis*, 20(1), 25–46. doi:10.1093/pan/mpr025

Zhao, Q., & Percival, D. (2017). Entropy balancing is doubly robust. *Journal of Causal Inference*, 5(1). doi:10.1515/jci20160010

`estimand = "ATE"`*:*

Källberg, D., & Waernbaum, I. (2023). Large Sample Properties of Entropy Balancing Estimators of Average Causal Effects. *Econometrics and Statistics*. doi:10.1016/j.ecosta.2023.11.004

Continuous Treatments:

Tübbicke, S. (2022). Entropy Balancing for Continuous Treatments. *Journal of Econometric Methods*, 11(1), 71–89. doi:10.1515/jem20210002

Vegetabile, B. G., Griffin, B. A., Coffman, D. L., Cefalu, M., Robbins, M. W., & McCaffrey, D. F. (2021). Nonparametric estimation of population average dose-response curves using entropy balancing weights for continuous exposures. *Health Services and Outcomes Research Methodology*, 21(1), 69–110. doi:10.1007/s10742020002362

See Also

`weightit()`, `weightitMSM()`

method_ipt and method_cbps for inverse probability tilting and CBPS, which work similarly. method_optweight for another implementation of entropy balancing (by setting `norm = "entropy"`).

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ebal", estimand = "ATT"))

summary(W1)

cobalt::bal.tab(W1)
```

```
#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ebal", estimand = "ATE"))

summary(W2)

cobalt::bal.tab(W2)

#Balancing covariates and squares with respect to
#re75 (continuous), maintaining 3 moments of the
#covariate and treatment distributions
(W3 <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ebal", moments = 2,
                d.moments = 3))

summary(W3)

cobalt::bal.tab(W3, poly = 2,
                stats = c("c", "m"))

#Balancing covariates between treatment groups (binary),
#allowing for inexact balance
(W1b <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ebal", estimand = "ATT",
                tols = .02))

summary(W1b)

cobalt::bal.tab(W1, weights = list(inexact = W1b))
```

`method_energy` *Energy Balancing*

Description

This page explains the details of estimating weights using energy balancing by setting `method = "energy"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating weights by minimizing an energy statistic related to covariate balance. For binary and multi-category treatments, this is the energy distance, which is a multivariate distance between distributions, between the treatment groups. For continuous treatments, this is the sum of the distance covariance between the treatment variable and the covariates and the energy distances between the treatment and covariates in the weighted sample and their distributions in the original sample. This method relies on code written for **WeightIt** using `osqp::osqp()` from the **osqp** package to perform the optimization. This method may be slow or memory-intensive for large datasets.

Binary Treatments:

For binary treatments, this method estimates the weights using `osqp()` using formulas described by Huling and Mak (2024). The following estimands are allowed: ATE, ATT, and ATC.

Multi-Category Treatments:

For multi-category treatments, this method estimates the weights using `osqp()` using formulas described by Huling and Mak (2024). The following estimands are allowed: ATE and ATT.

Continuous Treatments:

For continuous treatments, this method estimates the weights using `osqp()` using formulas described by Huling, Greifer, and Chen (2023).

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, the energy distance minimized is that between the weighted units still under observation and the full at-risk sample (censored and uncensored combined), rather than that between two treatment groups. Weights are estimated for the units still under observation only; the censored units receive a weight of 0. `improved` does not apply, as only one group is weighted.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point. This method is not guaranteed to yield optimal balance at each time point. **NOTE: the use of energy balancing with longitudinal treatments has not been validated!** Because of this, `weightitMSM()` errors when this method is requested; set `weightit.force = TRUE` to bypass that error.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios. In some cases, sampling weights will cause the optimization to fail due to lack of convexity or infeasible constraints.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Details

Energy balancing is a method of estimating weights using optimization without a propensity score. The weights are the solution to a constrained quadratic optimization problem where the objective function concerns covariate balance as measured by the energy distance and (for continuous treatments) the distance covariance.

Energy balancing for binary and multi-category treatments involves minimizing the energy distance between the treatment groups and between each treatment group and a target group (e.g., the full sample for the ATE). The energy distance is a scalar measure of the difference between two multivariate distributions and is equal to 0 when the two distributions are identical.

Energy balancing for continuous treatments involves minimizing the distance covariance between the treatment and the covariates; the distance covariance is a scalar measure of the association between two (possibly multivariate) distributions that is equal to 0 when the two distributions are independent. In addition, the energy distances between the treatment and covariate distributions in the weighted sample and the treatment and covariate distributions in the original sample are minimized.

The primary benefit of energy balancing is that all features of the covariate distribution are balanced, not just means, unlike with other optimization-based methods like entropy balancing. Still, it is possible to add additional balance constraints to require balance on individual terms using the `moments` argument, as with entropy balancing. Energy balancing can sometimes yield weights with high variability; the `lambda` argument can be supplied to penalize highly variable weights to increase the effective sample size at the expense of balance.

Reproducibility:

Although there are no stochastic components to the optimization, **osqp** by default updates the optimization based on how long the optimization has been running, which will vary across runs even when a seed is set and no parameters have been changed. See the discussion here for more details. To ensure reproducibility by default, `adaptive_rho_interval` is set to 50, which fixes the update schedule at a set number of iterations. See `osqp::osqpSettings()` for details.

Additional Arguments

The following additional arguments can be specified:

`dist.mat` the name of the method used to compute the distance matrix of the covariates or the numeric distance matrix itself. Allowable options include `"scaled_euclidean"` for the Euclidean (L2) distance on the scaled covariates (the default), `"mahalanobis"` for the Mahalanobis distance, and `"euclidean"` for the raw Euclidean distance. Abbreviations allowed. Note that some user-supplied distance matrices can cause the R session to abort due to a bug within **osqp**, so this argument should be used with caution. A distance matrix must be a square, symmetric, numeric matrix with zeros along the diagonal and a row and column for each unit. Can also be supplied as the output of a call to `dist()`.

`lambda` a positive numeric scalar used to penalize the square of the weights. This value divided by the square of the total sample size is added to the diagonal of the quadratic part of the loss function. Higher values favor weights with less variability. Note this is distinct from the lambda value described in Huling and Mak (2024), which penalizes the complexity of individual treatment rules rather than the weights, but does correspond to lambda from Huling et al. (2023). Default is .0001, which is essentially 0.

`moments integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 0 to impose no constraint on balance.

`int logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`tols` when `moments` is positive, a number corresponding to the maximum allowed standardized mean difference (for binary and multi-category treatments) or treatment-covariate correlation (for continuous treatments) allowed. Default is 0. Ignored when `moments = 0`.

`min.w` the minimum allowable weight. Negative values (including `-Inf`) are allowed. Default is `1e-8`.

For binary and multi-category treatments, the following additional arguments can be specified:

`improved logical`; whether to use the improved energy balancing weights as described by Huling and Mak (2024) when `estimand = "ATE"`. This involves optimizing balance not only between each treatment group and the overall sample, but also between each pair of treatment groups. Huling and Mak (2024) found that the improved energy balancing weights generally outperformed standard energy balancing. Default is `TRUE`; set to `FALSE` to use the standard energy balancing weights instead (not recommended).

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates.

For continuous treatments, the following additional arguments can be specified:

`d.moments` The number of moments of the treatment and covariate distributions that are constrained to be the same in the weighted sample as in the original sample. For example, setting `d.moments = 3` ensures that the mean, variance, and skew of the treatment and covariates are the same in the weighted sample as in the unweighted sample. `d.moments` should be greater than or equal to `moments` and will be automatically set accordingly if not (or if not specified).

`dimension.adj logical`; whether to include the dimensionality adjustment described by Huling et al. (2023). If `TRUE`, the default, the energy distance for the covariates is weighted $\sqrt{p}$ times as much as the energy distance for the treatment, where p is the number of covariates. If `FALSE`, the two energy distances are given equal weights. Default is `TRUE`.

`treat.dist.mat` the numeric distance matrix of the treatment, playing the same role for the treatment that `dist.mat` plays for the covariates. If unspecified, the Euclidean distance on the treatment scaled by its weighted standard deviation is used. Must be a square, symmetric, numeric matrix with zeros along the diagonal and a row and column for each unit; can also be supplied as the output of a call to `dist()`. As with `dist.mat`, some user-supplied matrices can cause the R session to abort due to a bug within **osqp**, so this argument should be used with caution.

The `moments` argument functions differently for `method = "energy"` from how it does with other methods. When unspecified or set to zero, energy balancing weights are estimated as described by Huling and Mak (2024) for binary and multi-category treatments or by Huling et al. (2023) for continuous treatments. When `moments` is set to an integer larger than 0, additional balance constraints on the requested moments of the covariates are also included, guaranteeing exact moment balance on these covariates while minimizing the energy distance of the weighted sample. For binary and multi-category treatments, this involves exact balance on the means of the entered covariates; for continuous treatments, this involves exact balance on the treatment-covariate correlations of the entered covariates. The constraint on exact balance can be relaxed using the `tols` argument.

Any other arguments will be passed to `osqp::osqpSettings()`. Some defaults differ from those in `osqpSettings()`; see *Reproducibility* section.

Additional Outputs

`obj` When `include.obj = TRUE`, the output of the call to `osqp::solve_osqp()`, which contains the dual variables and convergence information.

Note

Sometimes the optimization can fail to converge because the problem is not convex. A warning will be displayed if so. Because the optimization is deterministic (see the *Reproducibility* section above), re-fitting without changing anything will reproduce the failure; increasing `max_iter` or changing `adaptive_rho_interval` may help instead. If the method repeatedly fails, you should try another method or change the supplied parameters (though this is uncommon).

If it seems like the weights are balancing the covariates but you still get a failure to converge, this usually indicates that more iterations are needed to find the optimal solution. This can occur when `moments` or `int` are specified. `max_iter` should be increased, and setting `verbose = TRUE` allows you to monitor the process and examine if the optimization is approaching convergence.

If `min.w` is positive and you still get a warning about the presence of negative weights, try setting `eps` to a smaller number (e.g., to `1e-12`).

As of version 1.5.0, `polish` is now set to `TRUE` by default. This should yield slightly improved solutions but may be a little slower.

Author(s)

Noah Greifer, using code from Jared Huling's **independenceWeights** package for continuous treatments.

References

Binary and multi-category treatments:

Huling, J. D., & Mak, S. (2024). Energy balancing of covariate distributions. *Journal of Causal Inference*, 12(1). doi:10.1515/jci20220029

Continuous treatments:

Huling, J. D., Greifer, N., & Chen, G. (2023). Independence weights for causal inference with continuous treatments. *Journal of the American Statistical Association*, 0(ja), 1–25. doi:10.1080/01621459.2023.2213485

See Also

`weightit()`, `weightitMSM()`

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "energy", estimand = "ATE"))

summary(W1)

cobalt::bal.tab(W1)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "energy", estimand = "ATT",
                focal = "black"))

summary(W2)

cobalt::bal.tab(W2)

#Balancing covariates with respect to re75 (continuous)
(W3 <- weightit(re75 ~ age + educ + married +
                    nodegree + re74, data = lalonde,
                  method = "energy", moments = 1,
                  tols = .01))

summary(W3)

cobalt::bal.tab(W3, poly = 2)
```

---

method_gbm *Propensity Score Weighting Using Generalized Boosted Models*

Description

This page explains the details of estimating weights from generalized boosted model-based propensity scores by setting `method = "gbm"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating propensity scores using generalized boosted modeling (GBM) and then converting those propensity scores into weights using a formula that depends on the desired estimand. The algorithm involves using a balance-based or prediction-based criterion to optimize in choosing the value of tuning parameters (the number of trees and possibly others). The method relies on the **gbm** package.

This method mimics the functionality of functions in the **twang** package, but has improved performance and more flexible options. See Details section for more details.

Binary Treatments:

For binary treatments, this method estimates the propensity scores using `gbm::gbm.fit()` and then selects the optimal tuning parameter values using the method specified in the `criterion` argument. The following estimands are allowed: ATE, ATT, ATC, ATO, and ATM. The weights are computed from the estimated propensity scores using `get_w_from_ps()`, which implements the standard formulas. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Multi-Category Treatments:

For multi-category treatments, this method estimates the propensity scores using a single call to `gbm::gbm.fit()` with `distribution = "multinomial"` and then selects the optimal tuning parameter values using the method specified in the `criterion` argument. The following estimands are allowed: ATE, ATT, ATC, ATO, and ATM. The weights are computed from the estimated propensity scores using `get_w_from_ps()`, which implements the standard formulas. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Continuous Treatments:

For continuous treatments, weights are estimated as $w_i = f_A(a_i)/f_{A|X}(a_i)$, where $f_A(a_i)$ (known as the stabilization factor) is the unconditional density of treatment evaluated the observed treatment value and $f_{A|X}(a_i)$ (known as the generalized propensity score) is the conditional density of treatment given the covariates evaluated at the observed treatment value. The shape of $f_{A|X}(.)$ is controlled by the `density` argument described below (normal distribution by default), and the predicted values used for the mean of the conditional density are estimated using GBM as implemented in `gbm::gbm.fit()` with optimal tuning parameter values chosen using the method specified in the `criterion` argument. $f_A(.)$ is estimated by marginalizing over $f_{A|X}(.)$. Kernel density estimation can be used instead of assuming a specific density for the denominator by setting `density = "kernel"`. Other arguments to `density()` can be specified to refine the density estimation parameters.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, a single model of the probability of being censored is fit, and the weights are $1/P(C = 0|X)$ for the units still under observation and 0 for the censored units. The number of trees is selected using `criterion` computed between the *weighted units still under observation* and the *full at-risk sample*, which is what the weights target. This is cobalt's "target" balance, so `criterion` must be one of `cobalt::available.stats("target")` – a subset of the values allowed for binary treatments, excluding `"r2"`, `"r2.2"`, `"r2.3"`, `"kernel.dist"`, and `"l1.med"`. `"cv{#}"` works as it does for binary treatments.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** Missing variables are automatically handled by the boosting model. When using a `criterion` targeting covariate balance, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are included as covariates to balance. The missing values in the covariates are then replaced with the covariate medians. Balance assessment then proceeds with this new set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Parallelization:

Parallelization is supported; when a `future` plan is specified with more than one worker, balance is assessed on multiple trees in parallel.

Details

Generalized boosted modeling (GBM, also known as gradient boosting machines) is a machine learning method that generates predicted values from a flexible regression of the treatment on the covariates, which are treated as propensity scores and used to compute weights. It does this by building a series of regression trees, each fit to the residuals of the last, minimizing a loss function that depends on the distribution chosen. The optimal number of trees is a tuning parameter that must be chosen; McCaffrey et al. (2004) were innovative in using covariate balance to select this value rather than traditional machine learning performance metrics such as cross-validation accuracy. GBM is particularly effective for fitting nonlinear treatment models characterized by curves and interactions, but performs worse for simpler treatment models. It is unclear which balance measure should be used to select the number of trees, though research has indicated that balance measures tend to perform better than cross-validation accuracy for estimating effective propensity score weights.

WeightIt offers almost identical functionality to **twang**, the first package to implement this method. Compared to the current version of **twang**, **WeightIt** offers more options for the measure of balance used to select the number of trees, improved performance, tuning of hyperparameters, more estimands, and support for continuous treatments. **WeightIt** computes weights for multi-category treatments differently from how **twang** does; rather than fitting a separate binary GBM for each pair of treatments, **WeightIt** fits a single multi-class GBM model and uses balance measures appropriate for multi-category treatments.

`plot()` can be used on the output of `weightit()` with `method = "gbm"` to display the results of the tuning process; see Examples and `plot.weightit()` for more details.

Additional Arguments

The following additional arguments can be specified:

`criterion` A string describing the balance criterion used to select the best weights. See `cobalt::bal.compute()` for allowable options for each treatment type. In addition, to optimize the cross-validation error instead of balance, `criterion` can be set as `"cv{#}"`, where `{#}` is replaced by a number representing the number of cross-validation folds used (e.g., `"cv5"` for 5-fold cross-validation). For binary and multi-category treatments, the default is `"smd.mean"`, which minimizes the average absolute standard mean difference among the covariates between treatment

groups. For continuous treatments, the default is "p.mean", which minimizes the average absolute Pearson correlation between the treatment and covariates.

trim.at A number supplied to at in trim() which trims the weights from all the trees before choosing the best tree. This can be valuable when some weights are extreme, which occurs especially with continuous treatments. The default is 0 (i.e., no trimming).

subclass integer; the number of subclasses to use for computing weights using marginal mean weighting through stratification (MMWS). If NULL, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and weights will be computed based on subclass membership. Only allowed for binary and multi-category treatments. See get_w_from_ps() for details and references.

distribution A string with the distribution used in the loss function of the boosted model. This is supplied to the distribution argument in gbm::gbm.fit(). For binary treatments, "bernoulli" and "adaboost" are available, with "bernoulli" the default. For multi-category treatments, only "multinomial" is allowed. For continuous treatments "gaussian", "laplace", and "tdist" are available, with "gaussian" the default. This argument is tunable.

n.trees The maximum number of trees used. This is passed onto the n.trees argument in gbm.fit(). The default is 10000 for binary and multi-category treatments and 20000 for continuous treatments.

n.grid When criterion is a balance-based criterion, the number of points in the initial coarse grid of trees searched before the search is refined around the best point. Default is round(1 + sqrt(2 * (n.trees - start.tree + 1))). Must be between 2 and n.trees. Larger values make the initial search finer but slower.

n.cores When criterion is a cross-validation criterion (i.e., "cv{#}"), the number of cores used by gbm::gbmCrossVal() to fit the folds in parallel. Default is 1 for serial fitting.

start.tree The tree at which to start balance checking. If you know the best balance isn't in the first 100 trees, for example, you can set start.tree = 101 so that balance statistics are not computed on the first 100 trees. This can save some time since balance checking takes up the bulk of the run time for some balance-based stopping methods, and is especially useful when running the same model adding more and more trees. The default is 1, i.e., to start from the very first tree in assessing balance.

interaction.depth The depth of the trees. This is passed onto the interaction.depth argument in gbm.fit(). Higher values indicate better ability to capture nonlinear and nonadditive relationships. The default is 3 for binary and multi-category treatments and 4 for continuous treatments. This argument is tunable.

shrinkage The shrinkage parameter applied to the trees. This is passed onto the shrinkage argument in gbm.fit(). The default is .01 for binary and multi-category treatments and .0005 for continuous treatments. The lower this value is, the more trees one may have to include to reach the optimum. This argument is tunable.

bag.fraction The fraction of the units randomly selected to propose the next tree in the expansion. This is passed onto the bag.fraction argument in gbm.fit(). The default is 1, but smaller values should be tried. For values less than 1, subsequent runs with the same parameters will yield different results due to random sampling; be sure to set the seed using set.seed() to ensure replicability of results.

use.offset logical; whether to use the linear predictor resulting from a generalized linear model as an offset to the GBM model. If TRUE, this fits a logistic regression model (for binary

treatments) or a linear regression model (for continuous treatments) and supplies the linear predictor to the `offset` argument of `gbm.fit()`. This often improves performance generally but especially when the true propensity score model is well approximated by a GLM, and this yields uniformly superior performance over `method = "glm"` with respect to `criterion`. Default is `FALSE` to omit the offset. Only allowed for binary and continuous treatments. This argument is tunable.

All other arguments take on the defaults of those in `gbm::gbm.fit()`, and some are not used at all. For binary and multi-category treatments with cross-validation used as the criterion, `class.stratify.cv` is set to `TRUE` by default.

The `w` argument in `gbm.fit()` is ignored because sampling weights are passed using `s.weights`.

For continuous treatments only, the following arguments may be supplied:

`density` A function corresponding to the conditional density of the treatment. The standardized residuals of the treatment model will be fed through this function to produce the numerator and denominator of the generalized propensity score weights. This can also be supplied as a string containing the name of the function to be called. If the string contains underscores, the call will be split by the underscores and the latter splits will be supplied as arguments to the second argument and beyond. For example, if `density = "dt_2"` is specified, the density used will be that of a t-distribution with 2 degrees of freedom. Using a t-distribution can be useful when extreme outcome values are observed (Naimi et al., 2014).

Can also be `"kernel"` to use kernel density estimation, which calls `density()` to estimate the numerator and denominator densities for the weights. (This used to be requested by setting `use.kernel = TRUE`, which is now deprecated.)

If unspecified, a density corresponding to the argument passed to `distribution` is used. If `"gaussian"` (the default), `dnorm()` is used. If `"tdist"`, a t-distribution with 4 degrees of freedom is used. If `"laplace"`, a Laplace distribution is used.

`bw, adjust, kernel, n` If `density = "kernel"`, the arguments to `density()`. The defaults are the same as those in `density()`.

For tunable arguments, multiple entries may be supplied, and `weightit()` will choose the best value by optimizing the criterion specified in `criterion`. See below for additional outputs that are included when arguments are supplied to be tuned. See Examples for an example of tuning. The same seed is used for every run to ensure any variation in performance across tuning parameters is due to the specification and not to using a random seed. This only matters when `bag.fraction` differs from 1 (its default) or cross-validation is used as the criterion; otherwise, there are no random components in the model.

Additional Outputs

`info` A list with the following entries:

`best.tree` The number of trees at the optimum. If this is close to `n.trees`, `weightit()` should be rerun with a larger value for `n.trees`, and `start.tree` can be set to just below `best.tree`. When other parameters are tuned, this is the best tree value in the best combination of tuned parameters. See example.

`tree.val` A data frame with two columns: the first is the number of trees and the second is the value of the criterion corresponding to that tree. When other parameters are tuned, these are the number of trees and the criterion values in the best combination of tuned parameters. See example.

If any arguments are to be tuned (i.e., they have been supplied more than one value), the following two additional components are included in `info`:

`tune` A data frame with a column for each argument being tuned, the best value of the balance criterion for the given combination of parameters, and the number of trees at which the best value was reached.

`best.tune` A one-row data frame containing the values of the arguments being tuned that were ultimately selected to estimate the returned weights.

`obj` When `include.obj = TRUE`, the `gbm` fit used to generate the predicted values.

Note

The `criterion` argument used to be called `stop.method`, which is its name in **twang**. `stop.method` still works for backward compatibility. Additionally, the criteria formerly named as `"es.mean"`, `"es.max"`, and `"es.rms"` have been renamed to `"smd.mean"`, `"smd.max"`, and `"smd.rms"`. The former are used in **twang** and will still work with `weightit()` for backward compatibility.

Estimated propensity scores are trimmed to 10^{-8} and $1 - 10^{-8}$ to ensure balance statistics can be computed.

An additional `missing` argument, `"surr"`, used to be allowed; it is now set to `"ind"` with a warning.

References

Binary treatments:

McCaffrey, D. F., Ridgeway, G., & Morral, A. R. (2004). Propensity Score Estimation With Boosted Regression for Evaluating Causal Effects in Observational Studies. *Psychological Methods*, 9(4), 403–425. doi:10.1037/1082989X.9.4.403

Multi-Category Treatments:

McCaffrey, D. F., Griffin, B. A., Almirall, D., Slaughter, M. E., Ramchand, R., & Burgette, L. F. (2013). A Tutorial on Propensity Score Estimation for Multiple Treatments Using Generalized Boosted Models. *Statistics in Medicine*, 32(19), 3388–3414. doi:10.1002/sim.5753

Continuous treatments:

Zhu, Y., Coffman, D. L., & Ghosh, D. (2015). A Boosting Algorithm for Estimating Generalized Propensity Scores with Continuous Treatments. *Journal of Causal Inference*, 3(1). doi:10.1515/jci20140022

See Also

`weightit()`, `weightitMSM()`

`gbm::gbm.fit()` for the fitting function.

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
```

```
                method = "gbm", estimand = "ATE",
                criterion = "smd.max",
                use.offset = TRUE))

summary(W1)

cobalt::bal.tab(W1)

# View information about the fitting process
W1$info$best.tree #best tree

plot(W1) #plot of criterion value against number of trees

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "gbm", estimand = "ATT",
                focal = "hispan", criterion = "ks.mean"))

summary(W2)

cobalt::bal.tab(W2, stats = c("m", "ks"))

#Balancing covariates with respect to re75 (continuous)
(W3 <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "gbm", density = "kernel",
                criterion = "p.rms", trim.at = .97))

summary(W3)

cobalt::bal.tab(W3)

#Using a t(3) density and illustrating the search for
#more trees.
W4a <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "gbm", density = "dt_3",
                criterion = "p.max",
                n.trees = 10000)

W4a$info$best.tree #10000; optimum hasn't been found

plot(W4a) #decreasing at right edge

W4b <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "gbm", density = "dt_3",
                criterion = "p.max",
                start.tree = 10000,
                n.trees = 20000)

W4b$info$best.tree #13417; optimum has been found
```

```
plot(W4b) #increasing at right edge

cobalt::bal.tab(W4b)

#Tuning hyperparameters
(W5 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "gbm", estimand = "ATT",
                criterion = "ks.max",
                interaction.depth = 2:4,
                distribution = c("bernoulli", "adaboost")))

W5$info$tune

W5$info$best.tune #Best values of tuned parameters

plot(W5) #plot criterion values against number of trees

cobalt::bal.tab(W5, stats = c("m", "ks"))
```

`method_glm` *Propensity Score Weighting Using Generalized Linear Models*

Description

This page explains the details of estimating weights from generalized linear model-based propensity scores by setting `method = "glm"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating propensity scores with a parametric generalized linear model and then converting those propensity scores into weights using a formula that depends on the desired estimand. For binary and multi-category treatments, a binomial or multinomial regression model (logistic, by default) is used to estimate the propensity scores as the predicted probability of being in each treatment given the covariates. For ordinal treatments, an ordinal regression model is used to estimate generalized propensity scores. For continuous treatments, a generalized linear model is used to estimate generalized propensity scores as the conditional density of treatment given the covariates.

Binary Treatments:

For binary treatments, this method estimates the propensity scores using `glm()`. The following estimands are allowed: ATE, ATT, ATC, ATO, ATM, and ATOS. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Multi-Category Treatments:

For multi-category treatments, the propensity scores are estimated using multinomial regression from one of a few functions depending on the argument supplied to `multi.method` (see Additional Arguments below). The following estimands are allowed: ATE, ATT, ATC, ATO, and ATM. The weights for each estimand are computed using the standard formulas. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details. Ordinal treatments are treated exactly the same as non-ordinal multi-category treatments except that additional models are available to estimate the generalized propensity score (e.g., ordinal logistic regression).

Continuous Treatments:

For continuous treatments, weights are estimated as $w_i = f_A(a_i)/f_{A|X}(a_i)$, where $f_A(a_i)$ (known as the stabilization factor) is the unconditional density of treatment evaluated the observed treatment value and $f_{A|X}(a_i)$ (known as the generalized propensity score) is the conditional density of treatment given the covariates evaluated at the observed treatment value. The shape of $f_{A|X}(.)$ is controlled by the `density` argument described below (normal distribution by default), and the predicted values used for the mean of the conditional density are estimated using linear regression. $f_A(.)$ is estimated by marginalizing over $f_{A|X}(.)$. Kernel density estimation can be used instead of assuming a specific density for the denominator by setting `density = "kernel"`. Other arguments to `density()` can be specified to refine the density estimation parameters.

Multilevel Treatment Models:

When the model `formula` contains **lme4**-style random effects terms (e.g., `treat ~ x1 + x2 + (1 | school)`), a multilevel (mixed-effects) model is used to estimate the propensity scores. This can improve balance and overlap when units are clustered (e.g., patients within hospitals or students within schools). The grouping (and any random-slope) variables are taken from `data`. For binary treatments, the model is fit using `lme4::glmer()` with the requested `link` (`"logit"`, `"probit"`, or `"cloglog"`); for continuous treatments, using `lme4::lmer()` or `lme4::glmer()`; and for multi-category treatments, using `mclogit::mblogit()` with its `random` argument (i.e., `multi.method` is set to `"mclogit"`). The propensity scores are the predicted probabilities (or conditional densities) that include the estimated random effects, i.e., cluster-specific predictions. M-estimation is not available for these models; robust (`HC0`) or bootstrap standard errors should be used instead when estimating treatment effects (see the M-estimation section below and `glm_weightit()`).

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, a single binomial regression model of the probability of being censored is fit, and the weights are `1/P(C = 0 | X)` for the units still under observation and 0 for the censored units. All of the `link` options, `missing` options, and multilevel models described above apply unchanged, and the returned propensity score is the probability of *being censored*.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios except for multi-category treatments with `multi.method = "mnp"` and for binary and continuous treatments with `missing = "saem"` (see below). Warning messages may appear otherwise about non-integer successes, and these can be ignored.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

`"saem"` For binary treatments with `link = "logit"`, multi-category treatments (where one such model is fit per treatment level), or continuous treatments, a stochastic approximation version of the EM algorithm (SAEM) is used via the **misaem** package. No additional covariates are created. See Jiang et al. (2019) for information on this method. In some cases, this is a suitable alternative to multiple imputation.

M-estimation:

For binary treatments, M-estimation is supported when `link` is neither `"flic"` nor `"flac"` (see below). For multi-category treatments, M-estimation is supported when `multi.method` is `"weightit"` (the default) or `"glm"`. M-estimation is not supported when `subclass` is specified. For continuous treatments, M-estimation is supported when `density` is not `"kernel"`. The conditional treatment variance and unconditional treatment mean and variance are included as parameters to estimate, as these all go into calculation of the weights. For all treatment types, M-estimation is not supported when `missing = "saem"` or when the model `formula` includes random effects terms (i.e., a multilevel model is fit; see *Multilevel Treatment Models* above). See `glm_weightit()` and `vignette("estimating-effects")` for details. For longitudinal treatments, M-estimation is supported whenever the underlying methods are.

Additional Arguments

The following additional argument can be specified for all treatment types:

`quick` `logical`; whether to fit the model with `glm.fit()` rather than `glm()`. This skips construction of the model frame and returns a smaller fit object, which can be worthwhile when fitting many models (e.g., in a simulation) but means the object returned by `include.obj = TRUE` lacks the components that require it. Default is `FALSE`. Ignored when a fitting function other than `glm()` is used (e.g., with random effects terms or `missing = "saem"`).

For binary treatments, the following additional argument can be specified:

`link` the link used in the generalized linear model for the propensity scores. `link` can be any of those allowed by `binomial()` as well as `"loglog"` and `"clog"`. A `br.` prefix can be added (e.g., `"br.logit"`); this changes the fitting method to the bias-corrected generalized linear models implemented in the **brglm2** package. `link` can also be either `"flic"` or `"flac"` to fit the corresponding Firth corrected logistic regression models implemented in the **logistf** package; with these, arguments matching those of `logistf::logistf.control()` are passed to the fitting function as `control`, and a `modcontrol` list is passed to its `modcontrol` argument.

`subclass` `integer`; the number of subclasses to use for computing weights using marginal mean weighting through stratification (MMWS). If `NULL`, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and

weights will be computed based on subclass membership. See get_w_from_ps() for details and references.

For multi-category treatments, the following additional arguments can be specified:

multi.method the method used to estimate the generalized propensity scores. Allowable options include "weightit" (the default) to use multinomial logistic regression implemented in **WeightIt**, "glm" to use a series of binomial models using glm(), "mclogit" to use multinomial logistic regression as implemented in mclogit::mblogit(), and "mnp" to use Bayesian multinomial probit regression as implemented in MNP::MNP(). "weightit" and "mclogit" should give near-identical results, the main difference being increased robustness and customizability when using "mclogit" at the expense of not being able to use M-estimation to compute standard errors after weighting. For ordered treatments, allowable options include "weightit" (the default) to use ordinal regression implemented in **WeightIt** or "polr" to use ordinal regression implemented in MASS::polr(). Ignored when missing = "saem". Using the defaults allows for the use of M-estimation and requires no additional dependencies, but other packages may provide benefits such as speed and flexibility.

Bias-reduced models are requested by adding a br. prefix to link rather than by supplying a multi.method (see below). Because MASS::polr() cannot fit bias-reduced models, supplying a br. link with multi.method = "polr" uses the **WeightIt** implementation instead, which also makes M-estimation available.

link The link used in the multinomial, binomial, or ordered regression model for the generalized propensity scores depending on the argument supplied to multi.method. When multi.method = "glm", link can be any of those allowed by binomial() as well as "loglog" and "clog". When treatment is ordered and multi.method is "weightit" or "polr", link can be any of "logit", "probit", "loglog", "cloglog", or "cauchit". Otherwise, link should be "logit" or not specified.

For unordered treatments with multi.method = "weightit" and for ordered treatments with multi.method of "weightit" or "polr", a br. prefix can be added (e.g., "br.logit") to request mean bias reduction, i.e., to solve the bias-reducing adjusted score equations of Firth (1993) rather than the score equations. See multinom_weightit() and ordinal_weightit() for details. The estimates are always finite, even when the maximum likelihood estimates are not, which can help when the treatment groups are nearly separated. M-estimation remains available.

subclass integer; the number of subclasses to use for computing weights using marginal mean weighting through stratification (MMWS). If NULL, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and weights will be computed based on subclass membership. See get_w_from_ps() for details and references.

For continuous treatments, the following additional arguments may be supplied:

density A function corresponding to the conditional density of the treatment. The standardized residuals of the treatment model will be fed through this function to produce the denominator of the generalized propensity score weights. If blank, dnorm() is used as recommended by Robins et al. (2000). This can also be supplied as a string containing the name of the function to be called. If the string contains underscores, the call will be split by the underscores and the latter splits will be supplied as arguments to the second argument and beyond. For example, if density = "dt_2" is specified, the density used will be that of a t-distribution with 2 degrees

of freedom. Using a t-distribution can be useful when extreme outcome values are observed (Naimi et al., 2014).

Can also be "kernel" to use kernel density estimation, which calls density() to estimate the denominator density for the weights. (This used to be requested by setting use.kernel = TRUE, which is now deprecated.)

bw, adjust, kernel, n If density = "kernel", the arguments to density(). The defaults are the same as those in density().

link The link used to fit the linear model for the generalized propensity score. Can be any allowed by gaussian().

For binary treatments, the method argument in glm() is renamed to glm.method. This can be used to supply alternative fitting functions, such as those implemented in the **glm2** package; it is ignored for multi-category and continuous treatments. Arguments matching those of glm.control() are collected and passed to glm() as its control argument; a control list can also be supplied directly. Other arguments are not forwarded to glm().

In the presence of missing data with link = "logit" and missing = "saem", a control argument is passed to misaem::miss.glm(), and the method argument of misaem::predict.miss.glm() is supplied as saem.method. The same applies to misaem::miss.lm() and misaem::predict.miss.lm() for continuous treatments.

When the model formula includes random effects terms (see *Multilevel Treatment Models* above), the fit is handled by lme4::glmer() for binary treatments, mclogit::mblogit() for multi-category treatments, and lme4::lmer() or lme4::glmer() for continuous treatments. For multi-category treatments, random, estimator, dispersion, and groups are passed on to mblogit(); for continuous treatments, a control argument is passed on. Binary treatments accept no additional arguments on this path.

Additional Outputs

obj When include.obj = TRUE, the (generalized) propensity score model fit. For binary treatments, the output of the call to glm() or the requested fitting function. For multi-category treatments, the output of the call to the fitting function (or a list thereof if multi.method = "glm"). For continuous treatments, the output of the call to glm() for the predicted values in the denominator density. When the model formula includes random effects terms, the output of the call to lme4::glmer(), lme4::lmer(), or mclogit::mblogit().

References

Binary treatments:

- estimand = "ATO"

Li, F., Morgan, K. L., & Zaslavsky, A. M. (2018). Balancing covariates via propensity score weighting. *Journal of the American Statistical Association*, 113(521), 390–400. doi:10.1080/01621459.2016.1260466

- estimand = "ATM"

Li, L., & Greene, T. (2013). A Weighting Analogue to Pair Matching in Propensity Score Analysis. *The International Journal of Biostatistics*, 9(2). doi:10.1515/ijb20120030

- estimand = "ATOS"

Crump, R. K., Hotz, V. J., Imbens, G. W., & Mitnik, O. A. (2009). Dealing with limited overlap in estimation of average treatment effects. *Biometrika*, 96(1), 187–199. doi:10.1093/biomet/asn055

- Other estimands

Austin, P. C. (2011). An Introduction to Propensity Score Methods for Reducing the Effects of Confounding in Observational Studies. *Multivariate Behavioral Research*, 46(3), 399–424. doi:10.1080/00273171.2011.568786

- Marginal mean weighting through stratification

Hong, G. (2010). Marginal mean weighting through stratification: Adjustment for selection bias in multilevel data. *Journal of Educational and Behavioral Statistics*, 35(5), 499–531. doi:10.3102/1076998609359785

- Bias-reduced regression

Firth, D. (1993). Bias reduction of maximum likelihood estimates. *Biometrika*, 80(1), 27–38. doi:10.1093/biomet/80.1.27

For binary treatments, see also the references for the **brglm2** package, which does the fitting. For multi-category treatments, the fitting is done by **WeightIt**; see `multinom_weightit()` and `ordinal_weightit()`, which document the adjustments used.

- Firth corrected logistic regression

Puhr, R., Heinze, G., Nold, M., Lusa, L., & Geroldinger, A. (2017). Firth's logistic regression with rare events: Accurate effect estimates and predictions? *Statistics in Medicine*, 36(14), 2302–2317. doi:10.1002/sim.7273

- SAEM logistic regression for missing data

Jiang, W., Josse, J., & Lavielle, M. (2019). Logistic regression with missing covariates — Parameter estimation, model selection and prediction within a joint-modeling framework. *Computational Statistics & Data Analysis*, 106907. doi:10.1016/j.csda.2019.106907

Multi-Category Treatments:

- `estimand = "ATO"`

Li, F., & Li, F. (2019). Propensity score weighting for causal inference with multiple treatments. *The Annals of Applied Statistics*, 13(4), 2389–2415. doi:10.1214/19AOAS1282

- `estimand = "ATM"`

Yoshida, K., Hernández-Díaz, S., Solomon, D. H., Jackson, J. W., Gagne, J. J., Glynn, R. J., & Franklin, J. M. (2017). Matching weights to simultaneously compare three treatment groups: Comparison to three-way matching. *Epidemiology* (Cambridge, Mass.), 28(3), 387–395. doi:10.1097/EDE.0000000000000627

- Other estimands

McCaffrey, D. F., Griffin, B. A., Almirall, D., Slaughter, M. E., Ramchand, R., & Burgette, L. F. (2013). A Tutorial on Propensity Score Estimation for Multiple Treatments Using Generalized Boosted Models. *Statistics in Medicine*, 32(19), 3388–3414. doi:10.1002/sim.5753

- Marginal mean weighting through stratification

Hong, G. (2012). Marginal mean weighting through stratification: A generalized method for evaluating multivalued and multiple treatments with nonexperimental data. *Psychological Methods*, 17(1), 44–60. doi:10.1037/a0024918

Continuous treatments:

Robins, J. M., Hernán, M. Á., & Brumback, B. (2000). Marginal Structural Models and Causal Inference in Epidemiology. *Epidemiology*, 11(5), 550–560.

- Using non-normal conditional densities

Naimi, A. I., Moodie, E. E. M., Auger, N., & Kaufman, J. S. (2014). Constructing Inverse Probability Weights for Continuous Exposures: A Comparison of Methods. *Epidemiology*, 25(2), 292–299. doi:10.1097/EDE.0000000000000053

- SAEM linear regression for missing data

Jiang, W., Josse, J., & Lavielle, M. (2019). Logistic regression with missing covariates — Parameter estimation, model selection and prediction within a joint-modeling framework. *Computational Statistics & Data Analysis*, 106907. doi:10.1016/j.csda.2019.106907

See Also

`weightit()`, `weightitMSM()`, `get_w_from_ps()`

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "glm", estimand = "ATT",
                link = "probit"))

summary(W1)

bal.tab(W1)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "glm", estimand = "ATE"))

summary(W2)

bal.tab(W2)

#Balancing covariates with respect to re75 (continuous)
#with kernel density estimate
(W3 <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "glm", density = "kernel"))

summary(W3)

bal.tab(W3)
```

`method_ipt` *Inverse Probability Tilting*

Description

This page explains the details of estimating weights using inverse probability tilting by setting `method = "ipt"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary and multi-category treatments, as well as for estimating censoring weights.

In general, this method relies on estimating propensity scores using a modification of the usual generalized linear model score equations to enforce balance and then converting those propensity scores into weights using a formula that depends on the desired estimand. This method relies on code written for **WeightIt** using `rootSolve::multiroot()`.

Binary Treatments:

For binary treatments, this method estimates the weights using formulas described by Graham, Pinto, and Egel (2012). The following estimands are allowed: ATE, ATT, and ATC. When the ATE is requested, the optimization is run twice, once for each treatment group.

Multi-Category Treatments:

For multi-category treatments, this method estimates the weights using modifications of the formulas described by Graham, Pinto, and Egel (2012). The following estimands are allowed: ATE and ATT. When the ATE is requested, estimation is performed once for each treatment group. When the ATT is requested, estimation is performed once for each non-focal (i.e., control) group.

Continuous Treatments:

Inverse probability tilting is not compatible with continuous treatments.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, the censoring model is tilted so that the weighted covariate means of the units still under observation equal those of the full at-risk sample. Only one set of tilting parameters is solved for, rather than one per treatment group as for the ATE. The censored units receive a weight of 0.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point. This method is not guaranteed to yield exact balance at each time point. **NOTE: the use of inverse probability tilting with longitudinal treatments has not been validated!**

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is supported for all scenarios. See `glm_weightit()` and `vignette("estimating-effects")` for details.

Details

Inverse probability tilting (IPT) involves specifying estimating equations that fit the parameters of two or more generalized linear models with a modification that ensures exact balance on the covariate means. These estimating equations are solved, and the estimated parameters are used in the (generalized) propensity score, which is used to compute the weights. Conceptually and mathematically, IPT is very similar to entropy balancing and just-identified CBPS. For the ATT and ATC, entropy balancing, just-identified CBPS, and IPT will yield identical results. For the ATE or when `link` is specified as something other than `"logit"`, the three methods differ.

Treatment effect estimates for binary treatments are consistent if the true propensity score is a logistic regression or the outcome model is linear in the covariates and their interaction with treatments. For entropy balancing, this is only true for the ATT, and for just-identified CBPS, this is only true if there is no effect modification by covariates. In this way, IPT provides additional theoretical guarantees over the other two methods, though potentially with some cost in precision.

Additional Arguments

`moments` `integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 1 to balance covariate means.

`int` `logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates.

`link` the link used to determine the inverse link for computing the (generalized) propensity scores. Default is `"logit"`, which is used in the original description of the method by Graham, Pinto, and Egel (2012), but `"probit"`, `"cauchit"`, `"cloglog"`, `"loglog"`, `"log"`, `"clog"`, and `"softplus"` are also allowed. Note that negative weights are possible with these last two and they should be used with caution. An object of class `"link-glm"` can also be supplied. The argument is passed to `quasibinomial()`.

The `stabilize` argument is ignored.

Additional Outputs

`obj` When `include.obj = TRUE`, the output of the call to `rootSolve::multiroot()`, which contains the coefficient estimates and convergence information. For ATE fits or with multi-category treatments, a list of `rootSolve::multiroot()` outputs, one for each weighted group.

References

`estimand = "ATE"`:

Graham, B. S., De Xavier Pinto, C. C., & Egel, D. (2012). Inverse Probability Tilting for Moment Condition Models with Missing Data. *The Review of Economic Studies*, 79(3), 1053–1079. doi:10.1093/restud/rdr047

`estimand = "ATT"`:

Sant'Anna, P. H. C., & Zhao, J. (2020). Doubly robust difference-in-differences estimators. *Journal of Econometrics*, 219(1), 101–122. doi:10.1016/j.jeconom.2020.06.003

See Also

`weightit()`, `weightitMSM()`

method_ebal and method_cbps for entropy balancing and CBPS, which work similarly.

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ipt", estimand = "ATT"))

summary(W1)

cobalt::bal.tab(W1)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ipt", estimand = "ATE"))

summary(W2)

cobalt::bal.tab(W2)
```

`method_npcbps` *Nonparametric Covariate Balancing Propensity Score Weighting*

Description

This page explains the details of estimating weights from nonparametric covariate balancing propensity scores by setting `method = "npcbps"` in the call to `weightit()` or `weightitMSM()`. This method can be used with binary, multi-category, and continuous treatments.

In general, this method relies on estimating weights by maximizing the empirical likelihood of the data subject to balance constraints. This method relies on `CBPS::npCBPS()` from the **CBPS** package.

Binary Treatments:

For binary treatments, this method estimates the weights using `CBPS::npCBPS()`. The ATE is the only estimand allowed. The weights are taken from the output of the `npCBPS` fit object.

Multi-Category Treatments:

For multi-category treatments, this method estimates the weights using `CBPS::npCBPS()`. The ATE is the only estimand allowed. The weights are taken from the output of the `npCBPS` fit object.

Continuous Treatments:

For continuous treatments, this method estimates the weights using `CBPS::npCBPS()`. The weights are taken from the output of the `npCBPS` fit object.

Censoring Weights:

Censoring weights are not available for this method, because `npCBPS()` provides no way to target an external set of covariate means or to restrict estimation to one group. For empirical-likelihood-flavored censoring weights, use `method = "optweight"` with `norm = "log"`; for the entropy analogue, use `method = "ebal"`.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point. **NOTE: the use of npCBPS with longitudinal treatments has not been validated!** Because of this, `weightitMSM()` errors when this method is requested; set `weightit.force = TRUE` to bypass that error.

Sampling Weights:

Sampling weights are **not** supported with `method = "npcbps"`.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Details

Nonparametric CBPS involves the specification of a constrained optimization problem over the weights. The constraints correspond to covariate balance, and the loss function is the empirical likelihood of the data given the weights. npCBPS is similar to entropy balancing and will generally produce similar results. Because the optimization problem of npCBPS is not convex it can be slow to converge or not converge at all, so approximate balance is allowed instead using the `corprior` argument, which controls the average deviation from zero correlation between the treatment and covariates allowed.

Additional Arguments

`moments integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 1 to balance covariate means.

`int logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates. Only allowed with binary and multi-category treatments.

All arguments to `npCBPS()` can be passed through `weightit()` or `weightitMSM()`.

All arguments take on the defaults of those in `npCBPS()`.

Additional Outputs

`obj` When `include.obj = TRUE`, the nonparametric CB(G)PS model fit. The output of the call to `CBPS::npCBPS()`.

References

Fong, C., Hazlett, C., & Imai, K. (2018). Covariate balancing propensity score for a continuous treatment: Application to the efficacy of political advertisements. *The Annals of Applied Statistics*, 12(1), 156–177. doi:10.1214/17AOAS1101

See Also

`weightit()`, `weightitMSM()`, `method_cbps`

`method_optweight`, which can also be used to perform npCBPS by setting `norm = "log"`. In general, this `"optweight"` implementation is more stable and flexible.

`CBPS::npCBPS()` for the fitting function

Examples

```
# Examples take a long time to run
data("lalonde", package = "cobalt")

  #Balancing covariates between treatment groups (binary)
  (W1 <- weightit(treat ~ age + educ + married +
                    nodegree + re74, data = lalonde,
                  method = "npcbps", estimand = "ATE"))

  summary(W1)

  cobalt::bal.tab(W1)

  #Balancing covariates with respect to race (multi-category)
  (W2 <- weightit(race ~ age + educ + married +
                    nodegree + re74, data = lalonde,
                  method = "npcbps", estimand = "ATE"))

  summary(W2)

  cobalt::bal.tab(W2)
```

`method_optweight` *Stable Balancing Weights*

Description

This page explains the details of estimating stable balancing weights (also known as optimization-based weights) by setting `method = "optweight"` in the call to `weightit()`. This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating weights by solving a quadratic programming problem subject to approximate or exact balance constraints. This method relies on `optweight::optweight.fit()` from the **optweight** package.

Because `optweight::optweight()` offers finer control and uses the same syntax as `weightit()`, it is recommended that `optweight()` be used instead of `weightit()` with `method = "optweight"`.

Binary Treatments:

For binary treatments, this method estimates the weights using `optweight::optweight.fit()`. The following estimands are allowed: ATE, ATT, and ATC. The weights are taken from the output of the `optweight.fit` fit object.

Multi-Category Treatments:

For multi-category treatments, this method estimates the weights using `optweight::optweight.fit()`. The following estimands are allowed: ATE and ATT. The weights are taken from the output of the `optweight.fit` fit object.

Continuous Treatments:

For continuous treatments, this method estimates the weights using `optweight::optweight.fit()`. The weights are taken from the output of the `optweight.fit` fit object.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, `optweight::optweight.svy.fit()` is used rather than `optweight::optweight.fit()`. It solves exactly the required problem: finding weights for the units still under observation whose weighted covariate means equal those of the full at-risk sample, without estimating any weights for the censored units, which receive a weight of 0. `tols` are standardized using the units still under observation. Note the resulting weights have a mean of 1 among those units, whereas the other methods put them on the $1/P(C = 0|X)$ scale; the two differ by a constant factor. M-estimation is not supported.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point. This method is not guaranteed to yield exact balance at each time point. **NOTE: the use of stable balancing weights with longitudinal treatments has not been validated and should not be done!** Because of this, `weightitMSM()` errors when this method is requested; set `weightit.force = TRUE` to bypass that error.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Details

Stable balancing weights are weights that solve a constrained optimization problem, where the constraints correspond to covariate balance and the loss function is the variance (or other norm) of the weights. These weights maximize the effective sample size of the weighted sample subject to user-supplied balance constraints. An advantage of this method over entropy balancing is the ability to allow approximate, rather than exact, balance through the `tols` argument, which can increase precision even for slight relaxations of the constraints.

The function of the weights that is optimized can be changed using the `norm` argument. The default `norm = "l2"`, minimizes the variance of the weights (i.e., maximizes the ESS). `norm = "entropy"` minimizes the negative entropy of the weights and is equivalent to entropy balancing, though in this implementation, inexact balance is allowed. `norm = "log"` minimizes the sum of the negative logs

of the weights and is equivalent to nonparametric covariate balancing propensity score weighting (npCBPS). See `optweight::optweight.fit()` for the other allowed options to `norm` and other arguments.

`plot()` can be used on the output of `weightit()` with `method = "optweight"` to display the dual variables; see Examples and `plot.weightit()` for more details.

Additional Arguments

`base.weights` a vector of base weights, one for each unit. The estimated weights minimize the chosen divergence from the base weights rather than from a vector of 1s, subject to the balance constraints. These can be used to supply previously estimated weights so that the newly estimated weights retain some of the properties of the original weights. Sampling weights should not be passed to `base.weights` but can be included in a `weightit()` call that includes `s.weights`. Can also be supplied as `b.weights` or `base.weight`.

`moments` `integer`; the highest power of each covariate to be balanced. For example, if `moments = 3`, each covariate, its square, and its cube will be balanced. Can also be a named vector with a value for each covariate (e.g., `moments = c(x1 = 2, x2 = 4)`). Values greater than 1 for categorical covariates are ignored. Default is 1 to balance covariate means.

`int` `logical`; whether first-order interactions of the covariates are to be balanced. Default is `FALSE`.

`quantile` a named list of quantiles (values between 0 and 1) for each continuous covariate, which are used to create additional variables that when balanced ensure balance on the corresponding quantile of the variable. For example, setting `quantile = list(x1 = c(.25, .5, .75))` ensures the 25th, 50th, and 75th percentiles of `x1` in each treatment group will be balanced in the weighted sample. Can also be a single number (e.g., `.5`) or a vector (e.g., `c(.25, .5, .75)`) to request the same quantile(s) for all continuous covariates. Only allowed with binary and multi-category treatments.

All arguments to `optweight.fit()` can be passed through `weightit()` or `weightitMSM()`, with the following exception:

- `targets` cannot be used and is ignored.

All arguments take on the defaults of those in `optweight.fit()`.

Additional Outputs

`info` A list with one entry:

`duals` A data frame of dual variables for each balance constraint.

`obj` When `include.obj = TRUE`, the output of the call to `optweight::optweight.fit()`.

Note

The specification of `tols` differs between `weightit()` and `optweight()`. In `weightit()`, one tolerance value should be included per level of each factor variable, whereas in `optweight()`, all levels of a factor are given the same tolerance, and only one value needs to be supplied for a factor variable. Because of the potential for confusion and ambiguity, it is recommended to only supply one value for `tols` in `weightit()` that applies to all variables. For finer control, use `optweight()` directly.

Seriously, just use `optweight::optweight()`. The syntax is almost identical and it's compatible with **cobalt**, too.

References

Binary treatments:

Wang, Y., & Zubizarreta, J. R. (2020). Minimal dispersion approximately balancing weights: Asymptotic properties and practical considerations. *Biometrika*, 107(1), 93–105. doi:10.1093/biomet/asz050

Zubizarreta, J. R. (2015). Stable Weights that Balance Covariates for Estimation With Incomplete Outcome Data. *Journal of the American Statistical Association*, 110(511), 910–922. doi:10.1080/01621459.2015.1023805

Multi-Category Treatments:

de los Angeles Resa, M., & Zubizarreta, J. R. (2020). Direct and Stable Weight Adjustment in Non-Experimental Studies With Multivalued Treatments: Analysis of the Effect of an Earthquake on Post-Traumatic Stress. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 183(4), 1387–1410. doi:10.1111/rssa.12561

Continuous treatments:

Greifer, N. (2020). *Estimating Balancing Weights for Continuous Treatments Using Constrained Optimization*. doi:10.17615/DYSSB342

See Also

`weightit()`, `weightitMSM()`

`optweight::optweight.fit()` for the fitting function.

`method_ebal` for entropy balancing, which is a special case of stable balancing weights.

`method_npcbps` for npCBPS weighting, which is also a special case of stable balancing weights.

Examples

```
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + race +
                  nodegree + re74, data = lalonde,
                method = "optweight", estimand = "ATT",
                tols = 0))

summary(W1)

cobalt::bal.tab(W1)

plot(W1)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
```

```
                method = "optweight", estimand = "ATE",
                tols = .01))

summary(W2)

cobalt::bal.tab(W2)

plot(W2)

#Balancing covariates with respect to re75 (continuous)
(W3 <- weightit(re75 ~ age + educ + race +
                  nodegree + re74, data = lalonde,
                method = "optweight", tols = .02))

summary(W3)

cobalt::bal.tab(W3)

plot(W3)
```

`method_ps` *Weighting from Supplied Propensity Scores*

Description

This page explains the details of computing weights from propensity scores that have already been estimated, requested by supplying them to the `ps` argument of `weightit()` or `weightitMSM()` instead of setting `method`. This can be used with binary, multi-category, and continuous treatments, as well as with censoring.

No model is fit; the supplied scores are converted to weights exactly as `get_w_from_ps()` would, and the result is a full `weightit` object rather than a bare vector of weights. `formula` must still contain the treatment variable, but the covariates on the right hand side play no role in the computation. Any `method` supplied alongside `ps` is ignored unless it is a user-supplied function.

Because nothing is estimated, no M-estimation components are produced, and `glm_weightit()` will treat the weights as fixed.

Binary Treatments:

`ps` may be a numeric vector of the probability of being in the treated group, a one-column matrix or data frame of the same, or a two-column matrix or data frame with one column per treatment level. When two columns are supplied and their names match the treatment levels, those names determine which column is which; otherwise the columns are assumed to be in the order of the treatment levels. All estimands allowed by `get_w_from_ps()` are available: `"ATE"`, `"ATT"`, `"ATC"`, `"ATO"`, `"ATM"`, and `"ATOS"`.

Multi-Category Treatments:

`ps` may be a matrix or data frame with one column per treatment level, or a numeric vector (or one-column matrix) giving each unit's probability of being in the treatment group it is actually in. The estimands `"ATE"`, `"ATT"`, `"ATC"`, `"ATO"`, and `"ATM"` are available.

Continuous Treatments:

`ps` is interpreted as the conditional mean of the treatment (i.e., the fitted values of a treatment model) rather than as a probability. The generalized propensity score is formed by feeding the standardized residuals through the requested `density`, exactly as in `method_glm`.

Longitudinal Treatments:

For longitudinal treatments, `ps` is supplied as a list with one entry per time point, and the weights are the product of the time-specific weights.

Censoring Weights:

Wrapping the censoring indicator in `.cens()` requests inverse probability of censoring weights. `ps` is then the probability of *being censored*; the weights are `1/P(C = 0 | X)` for the units still under observation and exactly 0 for the censored units.

Sampling Weights:

Sampling weights are supported and are applied to the resulting weights, but they play no part in computing them, since no model is fit.

Additional Arguments

For binary and multi-category treatments, the following additional argument can be specified:

`subclass` `integer`; the number of subclasses to use for computing weights using marginal mean weighting through stratification (MMWS). If `NULL`, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and weights will be computed based on subclass membership. See `get_w_from_ps()` for details and references.

For continuous treatments, the following additional arguments may be supplied:

`density` A function corresponding to the conditional density of the treatment. The standardized residuals of the treatment will be fed through this function to produce the denominator of the generalized propensity score weights. If blank, `dnorm()` is used as recommended by Robins et al. (2000). This can also be supplied as a string containing the name of the function to be called. If the string contains underscores, the call will be split by the underscores and the latter splits will be supplied as arguments to the second argument and beyond. For example, if `density = "dt_2"` is specified, the density used will be that of a t-distribution with 2 degrees of freedom. Using a t-distribution can be useful when extreme treatment values are observed (Naimi et al., 2014).

Can also be `"kernel"` to use kernel density estimation, which calls `density()` to estimate the denominator density for the weights. (This used to be requested by setting `use.kernel = TRUE`, which is now deprecated.)

`bw`, `adjust`, `kernel`, `n` If `density = "kernel"`, the arguments to `density()`. The defaults are the same as those in `density()`.

Additional Outputs

`obj` No fit object is produced, since no model is fit; `include.obj` has no effect.

References

See get_w_from_ps() for references on each estimand and on marginal mean weighting through stratification.

Naimi, A. I., Moodie, E. E. M., Auger, N., & Kaufman, J. S. (2014). Constructing Inverse Probability Weights for Continuous Exposures: A Comparison of Methods. *Epidemiology*, 25(2), 292–299. doi:10.1097/EDE.0000000000000053

Robins, J. M., Hernán, M. Á., & Brumback, B. (2000). Marginal Structural Models and Causal Inference in Epidemiology. *Epidemiology*, 11(5), 550–560.

See Also

weightit(), weightitMSM()

get_w_from_ps(), which performs the same computation but returns only the weights

method_glm, for estimating the propensity scores within weightit() instead of supplying them

Examples

```
data("lalonde", package = "cobalt")

# Estimate the propensity score separately, then supply it
fit <- glm(treat ~ age + educ + married + nodegree + re74,
           data = lalonde, family = binomial)

(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                ps = fitted(fit), estimand = "ATT"))

summary(W1)

# Marginal mean weighting through stratification
(W2 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                ps = fitted(fit), estimand = "ATT",
                subclass = 20))

summary(W2)
```

---

method_super *Propensity Score Weighting Using SuperLearner*

Description

This page explains the details of estimating weights from SuperLearner-based propensity scores by setting method = "super" in the call to weightit() or weightitMSM(). This method can be used with binary, multi-category, and continuous treatments, as well as for estimating censoring weights.

In general, this method relies on estimating propensity scores using the SuperLearner algorithm for stacking predictions and then converting those propensity scores into weights using a formula that

depends on the desired estimand. For binary and multi-category treatments, one or more binary classification algorithms are used to estimate the propensity scores as the predicted probability of being in each treatment given the covariates. For continuous treatments, regression algorithms are used to estimate generalized propensity scores as the conditional density of treatment given the covariates. This method relies on `SuperLearner::SuperLearner()` from the **SuperLearner** package.

Binary Treatments:

For binary treatments, this method estimates the propensity scores using `SuperLearner::SuperLearner()`. The following estimands are allowed: ATE, ATT, ATC, ATO, ATM, and ATOS. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Multi-Category Treatments:

For multi-category treatments, the propensity scores are estimated using several calls to `SuperLearner::SuperLearner()`, one for each treatment group; the treatment probabilities are not normalized to sum to 1. The following estimands are allowed: ATE, ATT, ATC, ATO, and ATM. The weights for each estimand are computed using the standard formulas. Weights can also be computed using marginal mean weighting through stratification for the ATE, ATT, and ATC. See `get_w_from_ps()` for details.

Continuous Treatments:

For continuous treatments, weights are estimated as $w_i = f_A(a_i)/f_{A|X}(a_i)$, where $f_A(a_i)$ (known as the stabilization factor) is the unconditional density of treatment evaluated the observed treatment value and $f_{A|X}(a_i)$ (known as the generalized propensity score) is the conditional density of treatment given the covariates evaluated at the observed treatment value. The shape of $f_{A|X}(.)$ is controlled by the `density` argument described below (normal distribution by default), and the predicted values used for the mean of the conditional density are estimated using `SuperLearner::SuperLearner()`. $f_A(.)$ is estimated by marginalizing over $f_{A|X}(.)$. Kernel density estimation can be used instead of assuming a specific density for the denominator by setting `density = "kernel"`. Other arguments to `density()` can be specified to refine the density estimation parameters.

Censoring Weights:

For censoring weights, requested by wrapping the censoring indicator in `.cens()`, a single SuperLearner model of the probability of being censored is fit, and the weights are `1/P(C = 0 | X)` for the units still under observation and 0 for the censored units. With `SL.method = "method.balance"`, the library weights are chosen using balance between the *weighted units still under observation* and the *full at-risk sample*, which is what the weights target. This is cobalt's "target" balance; accordingly, `criterion` must be one of `cobalt::available.stats("target")`, a subset of the values allowed for binary treatments.

Longitudinal Treatments:

For longitudinal treatments, the weights are the product of the weights estimated at each time point.

Sampling Weights:

Sampling weights are supported through `s.weights` in all scenarios.

Missing Data:

In the presence of missing data, the following value(s) for `missing` are allowed:

`"ind"` **(default)** First, for each variable with missingness, a new missingness indicator variable is created which takes the value 1 if the original covariate is `NA` and 0 otherwise. The missingness indicators are added to the model formula as main effects. The missing values in the covariates are then replaced with the covariate medians (this value is arbitrary and does not affect estimation). The weight estimation then proceeds with this new formula and set of covariates. The covariates output in the resulting `weightit` object will be the original covariates with the `NA`s.

M-estimation:

M-estimation is not supported.

Details

SuperLearner works by fitting several machine learning models to the treatment and covariates and then taking a weighted combination of the generated predicted values to use as the propensity scores, which are then used to construct weights. The machine learning models used are supplied using the `SL.library` argument; the more models are supplied, the higher the chance of correctly modeling the propensity score. It is a good idea to include parametric models, flexible and tree-based models, and regularized models among the models selected. The predicted values are combined using the method supplied in the `SL.method` argument (which is nonnegative least squares by default). A benefit of SuperLearner is that, asymptotically, it is guaranteed to perform as well as or better than the best-performing method included in the library. Using Balance SuperLearner by setting `SL.method = "method.balance"` works by selecting the combination of predicted values that minimizes an imbalance measure.

Additional Arguments

`discrete` if `TRUE`, uses discrete SuperLearner, which simply selects the best performing method. Default `FALSE`, which finds the optimal combination of predictions for the libraries using `SL.method`.

An argument to `SL.library` **must** be supplied. To see a list of available entries, use `SuperLearner::listWrappers()`.

All arguments to `SuperLearner::SuperLearner()` can be passed through `weightit()` or `weightitMSM()`, with the following exceptions:

- `obsWeights` is ignored because sampling weights are passed using `s.weights`.
- `method` in `SuperLearner()` is replaced with the argument `SL.method` in `weightit()`.

For binary and multi-category treatments, the following arguments may be supplied:

`subclass` integer; the number of subclasses to use for computing weights using marginal mean weighting through stratification (MMWS). If `NULL`, standard inverse probability weights (and their extensions) will be computed; if a number greater than 1, subclasses will be formed and weights will be computed based on subclass membership. See `get_w_from_ps()` for details and references.

For continuous treatments, the following arguments may be supplied:

`density` A function corresponding to the conditional density of the treatment. The standardized residuals of the treatment model will be fed through this function to produce the denominator of the generalized propensity score weights. If blank, `dnorm()` is used as recommended by

Robins et al. (2000). This can also be supplied as a string containing the name of the function to be called. If the string contains underscores, the call will be split by the underscores and the latter splits will be supplied as arguments to the second argument and beyond. For example, if `density = "dt_2"` is specified, the density used will be that of a t-distribution with 2 degrees of freedom. Using a t-distribution can be useful when extreme outcome values are observed (Naimi et al., 2014).

Can also be `"kernel"` to use kernel density estimation, which calls `density()` to estimate the denominator density for the weights. (This used to be requested by setting `use.kernel = TRUE`, which is now deprecated.)

`bw, adjust, kernel, n` If `density = "kernel"`, the arguments to `density()`. The defaults are the same as those in `density()`.

Balance SuperLearner:

In addition to the methods allowed by `SuperLearner()`, one can specify `SL.method = "method.balance"` to use "Balance SuperLearner" as described by Pirracchio and Carone (2018), wherein covariate balance is used to choose the optimal combination of the predictions from the methods specified with `SL.library`. Coefficients are chosen (one for each prediction method) so that the weights generated from the weighted combination of the predictions optimize a balance criterion, which must be set with the `criterion` argument, described below.

`criterion` a string describing the balance criterion used to select the best weights. See `cobalt::bal.compute()` for allowable options for each treatment type. For binary and multi-category treatments, the default is `"smd.mean"`, which minimizes the average absolute standard mean difference among the covariates between treatment groups. For continuous treatments, the default is `"p.mean"`, which minimizes the average absolute Pearson correlation between the treatment and covariates.

Note that this implementation differs from that of Pirracchio and Carone (2018) in that here, balance is measured only on the terms included in the model formula (i.e., not their interactions unless specifically included), and balance results from a sample weighted using the estimated predicted values as propensity scores, not a sample matched using propensity score matching on the predicted values. Binary and continuous treatments are supported, but currently multi-category treatments are not.

Additional Outputs

`info` For binary and continuous treatments, a list with two entries, `coef` and `cvRisk`. For multi-category treatments, a list of lists with these two entries, one for each treatment level.

`coef` The coefficients in the linear combination of the predictions from each method in `SL.library`. Higher values indicate that the corresponding method plays a larger role in determining the resulting predicted value, and values close to zero indicate that the method plays little role in determining the predicted value. When `discrete = TRUE`, these correspond to the coefficients that would have been estimated had `discrete` been `FALSE`.

`cvRisk` The cross-validation risk for each method in `SL.library`. Higher values indicate that the method has worse cross-validation accuracy. When `SL.method = "method.balance"`, the sample weighted balance statistic requested with `criterion`. Higher values indicate worse balance.

`obj` When `include.obj = TRUE`, the SuperLearner fit(s) used to generate the predicted values. For binary and continuous treatments, the output of the call to `SuperLearner::SuperLearner()`. For multi-category treatments, a list of outputs to calls to `SuperLearner::SuperLearner()`.

Note

Some methods formerly available in **SuperLearner** are now in **SuperLearnerExtra**, which can be found on GitHub at https://github.com/ecpolley/SuperLearnerExtra.

The `criterion` argument used to be called `stop.method`, which is its name in **twang**. `stop.method` still works for backward compatibility. Additionally, the criteria formerly named as `"es.mean"`, `"es.max"`, and `"es.rms"` have been renamed to `"smd.mean"`, `"smd.max"`, and `"smd.rms"`. The former are used in **twang** and will still work with `weightit()` for backward compatibility.

As of version 1.2.0, the default behavior for binary and multi-category treatments is to stratify on the treatment when performing cross-validation to ensure all treatment groups are represented in cross-validation. To recover previous behavior, set `cvControl = list(stratifyCV = FALSE)`.

References

Binary treatments:

Pirracchio, R., Petersen, M. L., & van der Laan, M. (2015). Improving Propensity Score Estimators' Robustness to Model Misspecification Using Super Learner. *American Journal of Epidemiology*, 181(2), 108–119. doi:10.1093/aje/kwu253

Continuous treatments:

Kreif, N., Grieve, R., Díaz, I., & Harrison, D. (2015). Evaluation of the Effect of a Continuous Treatment: A Machine Learning Approach with an Application to Treatment for Traumatic Brain Injury. *Health Economics*, 24(9), 1213–1228. doi:10.1002/hec.3189

Balance SuperLearner (`SL.method = "method.balance"`):

Pirracchio, R., & Carone, M. (2018). The Balance Super Learner: A robust adaptation of the Super Learner to improve estimation of the average treatment effect in the treated based on propensity score matching. *Statistical Methods in Medical Research*, 27(8), 2504–2518. doi:10.1177/0962280216682055

See `method_glm` for additional references.

See Also

`weightit()`, `weightitMSM()`, `get_w_from_ps()`

Examples

```
data("lalonde", package = "cobalt")

#Note: for time, all examples use a small set of
#      learners. Many more should be added if
#      possible, including a variety of model
#      types (e.g., parametric, flexible, tree-
#      based, regularized, etc.)

# Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "super", estimand = "ATT",
                SL.library = c("SL.glm", "SL.stepAIC",
```

```
                                "SL.glm.interaction")))

summary(W1)

cobalt::bal.tab(W1)

# Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "super", estimand = "ATE",
                SL.library = c("SL.glm", "SL.stepAIC",
                               "SL.glm.interaction")))

summary(W2)

cobalt::bal.tab(W2)

# Balancing covariates with respect to re75 (continuous)
# assuming t(8) conditional density for treatment
(W3 <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "super", density = "dt_8",
                SL.library = c("SL.glm", "SL.ridge",
                               "SL.glm.interaction")))

summary(W3)

cobalt::bal.tab(W3)

# Balancing covariates between treatment groups (binary)
# using balance SuperLearner to minimize the maximum
# KS statistic
(W4 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "super", estimand = "ATT",
                SL.library = c("SL.glm", "SL.stepAIC",
                               "SL.lda"),
                SL.method = "method.balance",
                criterion = "ks.max"))

summary(W4)

cobalt::bal.tab(W4, stats = c("m", "ks"))
```

`method_user` *User-Defined Functions for Estimating Weights*

Description

This page explains the details of estimating weights using a user-defined function. The function must take in arguments that are passed to it by `weightit()` or `weightitMSM()` and return a vector of weights or a list containing the weights.

To supply a user-defined function, the function object should be entered directly to `method`; for example, for a function `fun`, `method = fun`.

Point Treatments:

The following arguments are automatically passed to the user-defined function, which should have named parameters corresponding to them:

- `treat`: a vector of treatment status for each unit. This comes directly from the left hand side of the formula passed to `weightit()` and so will have its type (e.g., numeric, factor, etc.), which may need to be converted.
- `covs`: a data frame of covariate values for each unit. This comes directly from the right hand side of the formula passed to `weightit()`. The covariates are processed so that all columns are numeric; all factor variables are split into dummies and all interactions are evaluated. All levels of factor variables are given dummies, so the matrix of the covariates is not full rank. Users can use `make_full_rank()`, which accepts a numeric matrix or data frame and removes columns to make it full rank, if a full rank covariate matrix is desired.
- `s.weights`: a numeric vector of sampling weights, one for each unit.
- `ps`: a numeric vector of propensity scores.
- `subset`: a logical vector the same length as `treat` that is `TRUE` for units to be included in the estimation and `FALSE` otherwise. This is used to subset the input objects when `by` is used. `treat`, `covs`, `s.weights`, and `ps`, if supplied, will already have been subsetted by `subset`.
- `estimand`: a character vector of length 1 containing the desired estimand. The characters will have been converted to uppercase. If "ATC" was supplied to estimand, `weightit()` sets `focal` to the control level (usually 0 or the lowest level of `treat`) and sets `estimand` to "ATT".
- `focal`: a character vector of length 1 containing the focal level of the treatment when the estimand is the ATT (or the ATC as detailed above). `weightit()` ensures the value of focal is a level of `treat`.
- `stabilize`: a logical vector of length 1. It is not processed by `weightit()` before it reaches the fitting function.
- `missing`: a character vector of length 1 containing the missingness method in effect, or `""` when no missing data are present. See the `missing` argument of `weightit()`.
- `verbose`: a logical vector of length 1 indicating whether the fitting function should print additional output.

None of these parameters are required to be in the fitting function. These are simply those that are automatically available.

In addition, any additional arguments supplied to `weightit()` will be passed on to the fitting function. `weightit()` ensures the arguments correspond to the parameters of the fitting function and throws an error if an incorrectly named argument is supplied and the fitting function doesn't include `...` as a parameter.

The fitting function must output either a numeric vector of weights or a list (or list-like object) with an entry named either "w" or "weights". If a list, the list can contain other named entries, but only entries named "w", "weights", "ps", and "fit.obj" will be processed. "ps" is a vector

of propensity scores and "fit.obj" should be an object used in the fitting process that a user may want to examine and that is included in the `weightit` output object as "obj" when `include.obj = TRUE`. The "ps" and "fit.obj" components are optional, but "weights" or "w" is required.

Longitudinal Treatments:

Longitudinal treatments can be handled either by running the fitting function for point treatments for each time point and multiplying the resulting weights together or by running a method that accommodates multiple time points and outputs a single set of weights. For the former, `weightitMSM()` can be used with the user-defined function just as it is with `weightit()`. The latter method is not yet accommodated by `weightitMSM()`, but will be someday, maybe.

See Also

`weightit()`, `weightitMSM()`

Examples

```
data("lalonde", package = "cobalt")

#A user-defined version of method = "ps"
my.ps <- function(treat, covs, estimand, focal = NULL, ...) {
  covs <- make_full_rank(covs)
  d <- data.frame(treat, covs)
  f <- formula(d)
  ps <- glm(f, data = d, family = "binomial")$fitted
  w <- get_w_from_ps(ps, treat = treat, estimand = estimand,
                     focal = focal)

  list(w = w, ps = ps)
}

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = my.ps, estimand = "ATT"))
summary(W1)
cobalt::bal.tab(W1)

data("msmdata")
(W2 <- weightitMSM(list(A_1 ~ X1_0 + X2_0,
                        A_2 ~ X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0,
                        A_3 ~ X1_2 + X2_2 +
                          A_2 + X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0),
                   data = msmdata,
                   method = my.ps))

summary(W2)
cobalt::bal.tab(W2)

# Kernel balancing using the `kbal` package, available
```

```
# using `pak::pak("chadhazlett/KBAL")`.
# Only the ATT and ATC are available.

## Not run:
  kbal.fun <- function(treat, covs, estimand, focal, verbose, ...) {
    args <- list(...)

    if (!estimand %in% c("ATT", "ATC")) {
      stop('`estimand` must be "ATT" or "ATC".', call. = FALSE)
    }

    treat <- as.numeric(treat == focal)

    args <- args[names(args) %in% names(formals(kbal::kbal))]
    args$allx <- covs
    args$treatment <- treat
    args$printprogress <- verbose

    cat_cols <- apply(covs, 2L, function(x) length(unique(x)) <= 2)

    if (all(cat_cols)) {
      args$cat_data <- TRUE
      args$mixed_data <- FALSE
      args$scale_data <- FALSE
      args$linkernel <- FALSE
      args$drop_MC <- FALSE
    }
    else if (any(cat_cols)) {
      args$cat_data <- FALSE
      args$mixed_data <- TRUE
      args$cat_columns <- colnames(covs)[cat_cols]
      args$allx[,!cat_cols] <- scale(args$allx[,!cat_cols])
      args$cont_scale <- 1
    }
    else {
      args$cat_data <- FALSE
      args$mixed_data <- FALSE
    }

    k.out <- do.call(kbal::kbal, args)
    w <- k.out$w

    list(w = w, fit.obj = k.out)
  }

  (Wk <- weightit(treat ~ age + educ + married +
                    nodegree + re74, data = lalonde,
                  method = kbal.fun, estimand = "ATT",
                  include.obj = TRUE))
  summary(Wk)
  cobalt::bal.tab(Wk, stats = c("m", "ks"))

## End(Not run)
```

`msmdata` *Simulated data for a 3 time point sequential study*

Description

This is a simulated dataset of 7500 units with covariates and treatment measured three times and the outcome measured at the end from a hypothetical observational study examining the effect of treatment delivered at each time point on an adverse event.

The data were generated using a simple simulation mechanism. For further details on how the dataset was built, see the code at data-raw/msmdata.R.

The dataset is provided to illustrate the features of `weightitMSM()` and is not based on a realistic data-generating process, so it should not be used as a benchmark.

For simulating realistic data with a known data-generating mechanism, consider using the **simcausal** package.

Usage

```
msmdata
```

Format

A data frame with 7500 observations on the following 10 variables.

`X1_0` a count covariate measured at baseline

`X2_0` a binary covariate measured at baseline

`A_1` a binary indicator of treatment status at the first time point

`X1_1` a count covariate measured at the first time point (after the first treatment)

`X2_1` a binary covariate measured at the first time point (after the first treatment)

`A_2` a binary indicator of treatment status at the second time point

`X1_2` a count covariate measured at the second time point (after the second treatment)

`X2_2` a binary covariate measured at the second time point (after the second treatment)

`A_3` a binary indicator of treatment status at the third time point

`Y_B` a binary indicator of the outcome event (e.g., death)

Examples

```
data("msmdata")
```

multinom_weightit *Fitting (Weighted) Multinomial Regression Models*

Description

multinom_weightit() fits a multinomial logistic regression model with a covariance matrix that accounts for estimation of weights, if supplied. By default, this function uses M-estimation to construct a robust covariance matrix using the estimating equations for the weighting model and the outcome model when available.

Usage

```
multinom_weightit(
  formula,
  data,
  link = "logit",
  weightit = NULL,
  vcov = NULL,
  cluster,
  R = 500L,
  offset,
  start = NULL,
  control = list(...),
  x = FALSE,
  y = TRUE,
  contrasts = NULL,
  fwb.args = list(),
  br = FALSE,
  ...
)
```

Arguments

formula	an object of class formula (or one that can be coerced to that class): a symbolic description of the model to be fitted.
data	a data frame containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which the function is called.
link	a string corresponding to the desired link function. Currently, only "logit" is allowed.
weightit	a weightit or weightitMSM object; the output of a call to weightit() or weightitMSM(). If not supplied, an unweighted model will be fit.
vcov	string; the method used to compute the variance of the estimated parameters. Allowable options include "asympt", which uses the asymptotically correct M-estimation-based method that accounts for estimation of the weights when available; "const", which uses the usual maximum likelihood estimates (only

available when weightit is not supplied); "HC0", which computes the robust sandwich variance treating weights (if supplied) as fixed; "BS", which uses the traditional bootstrap (including re-estimation of the weights, if supplied); "FWB", which uses the fractional weighted bootstrap as implemented in fwb::fwb() (including re-estimation of the weights, if supplied); and "none" to omit calculation of a variance matrix. If NULL (the default), will use "asympt" if weightit is supplied and M-estimation is available and "HC0" otherwise. See the vcov_type component of the outcome object to see which was used.

cluster — optional; for computing a cluster-robust variance matrix, a variable indicating the clustering of observations, a list (or data frame) thereof, or a one-sided formula specifying which variable(s) from the fitted model should be used. Note the cluster-robust variance matrix uses a correction for small samples, as is done in sandwich::vcovCL() by default. Cluster-robust variance calculations are available only when vcov is "asympt", "HC0", "BS", or "FWB".

R — the number of bootstrap replications when vcov is "BS" or "FWB". Default is 500. Ignored otherwise.

offset — optional; a numeric vector containing the model offset. See offset(). An offset can also be present in the model formula.

start — optional starting values for the coefficients.

control — a list of parameters for controlling the fitting process.

x, y — logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value.

contrasts — an optional list defining contrasts for factor variables. See model.matrix().

fwb.args — an optional list of further arguments to supply to fwb::fwb() when vcov = "FWB".

br — logical; whether to use mean bias reduction, i.e., to solve the bias-reducing adjusted score equations of Firth (1993) rather than the score equations. This yields estimates with smaller asymptotic bias that are always finite, even when the maximum likelihood estimates are not (e.g., under separation). Default is FALSE. See Details.

... — arguments to be used to form the default control argument if it is not supplied directly.

Details

multinom_weightit() implements multinomial logistic regression using a custom function in **WeightIt** that optionally computes a coefficient variance matrix that can be adjusted to account for estimation of the weights if a weightit or weightitMSM object is supplied to the weightit argument. This implementation is less robust to failures than other multinomial logistic regression solvers and should be used with caution. Unless br = TRUE, estimation of coefficients should align with that from mlogit::mlogit() and mclogit::mblogit() but might differ from nnet::multinom() due to the relaxed convergence thresholds of the latter.

When no argument is supplied to weightit or there is no "Mparts" attribute in the supplied object, the default variance matrix returned will be the "HC0" sandwich variance matrix, which is robust to misspecification of the outcome family (including heteroscedasticity). Otherwise, the default variance matrix uses M-estimation to additionally adjust for estimation of the weights. When

possible, this often yields smaller (and more accurate) standard errors. See the individual methods pages to see whether and when an "Mparts" attribute is included in the supplied object. To request that a variance matrix be computed that doesn't account for estimation of the weights even when a compatible weightit object is supplied, set vcov = "HC0", which treats the weights as fixed.

Bootstrapping can also be used to compute the coefficient variance matrix; when vcov = "BS" or vcov = "FWB", which implement the traditional resampling-based and fractional weighted bootstrap, respectively, the entire process of estimating the weights and fitting the outcome model is repeated in bootstrap samples (if a weightit object is supplied). This accounts for estimation of the weights and can be used with any weighting method. It is important to set a seed using set.seed() to ensure replicability of the results. The fractional weighted bootstrap is more reliable but requires the weighting method to accept sampling weights (which most do, and you'll get an error if it doesn't). Setting vcov = "FWB" and supplying fwb.args = list(wtype = "multinom") also performs the resampling-based bootstrap but with the additional features **fwb** provides (e.g., a progress bar and parallelization).

Bias reduction:

When br = TRUE, the coefficients solve the bias-reducing adjusted score equations of Firth (1993) instead of the score equations, which removes the first-order term in the asymptotic bias of the estimates. Because the multinomial logistic regression model uses the canonical link, the adjusted score function is the gradient of the log-likelihood penalized by the Jeffreys invariant prior, $\log|F(\beta)|/2$, where $F(\beta)$ is the expected information matrix (Kosmidis & Firth, 2011). Consequently, the estimates are always finite and correspond to the maximum of a penalized likelihood. Estimation should align with that from brglm2::brmultinom() with its default type = "AS_mean".

Weights are treated as multinomial totals, as they are by **brglm2**, which makes the estimates invariant to whether the data are supplied as individual units or as groups of identical units with weights equal to their counts. As for br = TRUE in glm_weightit(), the reported variance matrix uses the information matrix at the estimates rather than the Jacobian of the adjusted score, i.e., the adjustment is treated as fixed; the two differ by a term that vanishes asymptotically. M-estimation and bootstrapping can be used with br = TRUE just as they can without it.

Value

A multinom_weightit object.

Unless vcov = "none", the vcov component contains the covariance matrix adjusted for the estimation of the weights if requested and a compatible weightit object was supplied. The vcov_type component contains the type of variance matrix requested. If cluster is supplied, it will be stored in the "cluster" attribute of the output object, even if not used.

The model component of the output object (also the model.frame() output) will include two extra columns when weightit is supplied: (weights) containing the weights used in the model (the product of the estimated weights and the sampling weights, if any) and (s.weights) containing the sampling weights, which will be 1 if s.weights is not supplied in the original weightit() call.

References

Firth, D. (1993). Bias reduction of maximum likelihood estimates. *Biometrika*, 80(1), 27–38. doi:10.1093/biomet/80.1.27

Kosmidis, I., & Firth, D. (2011). Multinomial logit bias reduction via the Poisson log-linear model. *Biometrika*, 98(3), 755–759. doi:10.1093/biomet/asr026

See Also

- `glm_weightit()` for fitting generalized linear models that adjust for estimation of the weights.
- `ordinal_weightit()` for fitting ordinal regression models that adjust for estimation of the weights.
- `coxph_weightit()` for fitting Cox proportional hazards models that adjust for estimation of the weights.
- `mclogit::mblogit()` for fitting multinomial regression models that do not account for estimation of the weights.
- `brglm2::brmultinom()` for fitting bias-reduced multinomial regression models that do not account for estimation of the weights.

Examples

```
data("lalonde", package = "cobalt")

# Logistic regression ATT weights
w.out <- weightit(treat ~ age + educ + married + re74,
                  data = lalonde, method = "glm",
                  estimand = "ATT")

# Multinomial logistic regression outcome model
# that adjusts for estimation of weights
lalonde$re78_3 <- factor(findInterval(lalonde$re78,
                                      c(0, 5e3, 1e4)))

fit <- multinom_weightit(re78_3 ~ treat,
                         data = lalonde,
                         weightit = w.out)

summary(fit)

# Same model using mean bias reduction
fit_br <- multinom_weightit(re78_3 ~ treat,
                            data = lalonde,
                            weightit = w.out,
                            br = TRUE)

summary(fit_br)
```

`ordinal_weightit` *Fitting (Weighted) Ordinal Regression Models*

Description

ordinal_weightit() fits an ordinal regression model with a covariance matrix that accounts for estimation of weights, if supplied. By default, this function uses M-estimation to construct a robust covariance matrix using the estimating equations for the weighting model and the outcome model when available.

Usage

```
ordinal_weightit(
  formula,
  data,
  link = "logit",
  weightit = NULL,
  vcov = NULL,
  cluster,
  R = 500L,
  offset,
  start = NULL,
  control = list(...),
  x = FALSE,
  y = TRUE,
  contrasts = NULL,
  fwb.args = list(),
  br = FALSE,
  ...
)
```

Arguments

formula	an object of class formula (or one that can be coerced to that class): a symbolic description of the model to be fitted.
data	a data frame containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which the function is called.
link	a string corresponding to the desired link function. Allowable options include "logit", "probit", "loglog", "cloglog", and "cauchit". Default is "logit" for ordinal logistic regression.
weightit	a weightit or weightitMSM object; the output of a call to weightit() or weightitMSM(). If not supplied, an unweighted model will be fit.
vcov	string; the method used to compute the variance of the estimated parameters. Allowable options include "asympt", which uses the asymptotically correct M-estimation-based method that accounts for estimation of the weights when available; "const", which uses the usual maximum likelihood estimates (only available when weightit is not supplied); "HC0", which computes the robust sandwich variance treating weights (if supplied) as fixed; "BS", which uses the traditional bootstrap (including re-estimation of the weights, if supplied); "FWB",

which uses the fractional weighted bootstrap as implemented in `fwb::fwb()` (including re-estimation of the weights, if supplied); and `"none"` to omit calculation of a variance matrix. If `NULL` (the default), will use `"asympt"` if `weightit` is supplied and M-estimation is available and `"HC0"` otherwise. See the `vcov_type` component of the outcome object to see which was used.

`cluster` optional; for computing a cluster-robust variance matrix, a variable indicating the clustering of observations, a list (or data frame) thereof, or a one-sided formula specifying which variable(s) from the fitted model should be used. Note the cluster-robust variance matrix uses a correction for small samples, as is done in `sandwich::vcovCL()` by default. Cluster-robust variance calculations are available only when `vcov` is `"asympt"`, `"HC0"`, `"BS"`, or `"FWB"`.

`R` the number of bootstrap replications when `vcov` is `"BS"` or `"FWB"`. Default is 500. Ignored otherwise.

`offset` optional; a numeric vector containing the model offset. See `offset()`. An offset can also be present in the model formula.

`start` optional starting values for the coefficients.

`control` a list of parameters for controlling the fitting process.

`x, y` logical values indicating whether the response vector and model matrix used in the fitting process should be returned as components of the returned value.

`contrasts` an optional list defining contrasts for factor variables. See `model.matrix()`.

`fwb.args` an optional list of further arguments to supply to `fwb::fwb()` when `vcov = "FWB"`.

`br` `logical`; whether to use mean bias reduction, i.e., to solve the bias-reducing adjusted score equations of Kosmidis (2014) rather than the score equations. This yields estimates with smaller asymptotic bias that are always finite, even when the maximum likelihood estimates are not (e.g., when an end category is unobserved). Default is `FALSE`. See Details.

`...` arguments to be used to form the default control argument if it is not supplied directly.

Details

`ordinal_weightit()` implements proportional odds ordinal regression using a custom function in **WeightIt** that optionally computes a coefficient variance matrix that can be adjusted to account for estimation of the weights if a `weightit` or `weightitMSM` object is supplied to the `weightit` argument. Estimation of coefficients should align with that from `MASS::polr()` unless `br = TRUE`.

When no argument is supplied to `weightit` or there is no `"Mparts"` attribute in the supplied object, the default variance matrix returned will be the "HC0" sandwich variance matrix, which is robust to misspecification of the outcome family (including heteroscedasticity). Otherwise, the default variance matrix uses M-estimation to additionally adjust for estimation of the weights. When possible, this often yields smaller (and more accurate) standard errors. See the individual methods pages to see whether and when an `"Mparts"` attribute is included in the supplied object. To request that a variance matrix be computed that doesn't account for estimation of the weights even when a compatible `weightit` object is supplied, set `vcov = "HC0"`, which treats the weights as fixed.

Bootstrapping can also be used to compute the coefficient variance matrix; when `vcov = "BS"` or `vcov = "FWB"`, which implement the traditional resampling-based and fractional weighted bootstrap,

respectively, the entire process of estimating the weights and fitting the outcome model is repeated in bootstrap samples (if a `weightit` object is supplied). This accounts for estimation of the weights and can be used with any weighting method. It is important to set a seed using `set.seed()` to ensure replicability of the results. The fractional weighted bootstrap is more reliable but requires the weighting method to accept sampling weights (which most do, and you'll get an error if it doesn't). Setting `vcov = "FWB"` and supplying `fwb.args = list(wtype = "multinom")` also performs the resampling-based bootstrap but with the additional features **fwb** provides (e.g., a progress bar and parallelization).

Bias reduction:

When `br = TRUE`, the coefficients solve the bias-reducing adjusted score equations for cumulative link models derived by Kosmidis (2014) instead of the score equations, which removes the first-order term in the asymptotic bias of the estimates. Reduction of bias is equivalent to a parameter-dependent additive adjustment of the multinomial counts, and the resulting estimates are always finite and remain invariant to linear transformations of the parameters and to reversal of the order of the response categories. In the special case of two response categories, this reduces to the bias-reduced generalized linear model available with `br = TRUE` in `glm_weightit()`; for a saturated model without covariates and a logit link, it amounts to adding $1/2$ to the counts of the first and last categories.

The adjusted score equations are solved by quasi-Fisher scoring started at the maximum likelihood estimates. Two components of `control` govern this: `br.maxit` (the maximum number of iterations, default 100) and `br.tol` (the convergence tolerance for the adjusted score relative to the sum of the weights, default `1e-10`).

Weights are treated as multinomial totals, which makes the estimates invariant to whether the data are supplied as individual units or as groups of identical units with weights equal to their counts. As for `br = TRUE` in `glm_weightit()`, the reported variance matrix uses the information matrix at the estimates rather than the Jacobian of the adjusted score, i.e., the adjustment is treated as fixed; the two differ by a term that vanishes asymptotically. M-estimation and bootstrapping can be used with `br = TRUE` just as they can without it.

Note that `brglm2::bracl()`, which also performs bias reduction for ordinal responses, fits an adjacent category logit model rather than a cumulative link model, so its estimates are not comparable to those from `ordinal_weightit()`. This is true of `parallel = TRUE` as well, which constrains the adjacent category logits rather than the cumulative ones to be parallel and so yields different fitted probabilities.

Value

An `ordinal_weightit` object.

Unless `vcov = "none"`, the `vcov` component contains the covariance matrix adjusted for the estimation of the weights if requested and a compatible `weightit` object was supplied. The `vcov_type` component contains the type of variance matrix requested. If `cluster` is supplied, it will be stored in the `"cluster"` attribute of the output object, even if not used.

The `model` component of the output object (also the `model.frame()` output) will include two extra columns when `weightit` is supplied: `(weights)` containing the weights used in the model (the product of the estimated weights and the sampling weights, if any) and `(s.weights)` containing the sampling weights, which will be 1 if `s.weights` is not supplied in the original `weightit()` call.

References

Firth, D. (1993). Bias reduction of maximum likelihood estimates. *Biometrika*, 80(1), 27–38. doi:10.1093/biomet/80.1.27

Kosmidis, I. (2014). Improved estimation in cumulative link models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 76(1), 169–196. doi:10.1111/rssb.12025

See Also

- `glm_weightit()` for fitting generalized linear models that adjust for estimation of the weights.
- `multinom_weightit()` for fitting multinomial regression models that adjust for estimation of the weights.
- `coxph_weightit()` for fitting Cox proportional hazards models that adjust for estimation of the weights.
- `MASS::polr()` for fitting ordinal regression models that do not account for estimation of the weights.

Examples

```
data("lalonde", package = "cobalt")

# Logistic regression ATT weights
w.out <- weightit(treat ~ age + educ + married + re74,
                  data = lalonde, method = "glm",
                  estimand = "ATT")

lalonde$re78_3o <- factor(findInterval(lalonde$re78,
                                       c(0, 5e3, 1e4)),
                          ordered = TRUE)


# Ordinal probit regression that adjusts for estimation
# of weights
fit <- ordinal_weightit(re78_3o ~ treat,
                        data = lalonde,
                        link = "probit",
                        weightit = w.out)

summary(fit)

# Same model using mean bias reduction
fit_br <- ordinal_weightit(re78_3o ~ treat,
                           data = lalonde,
                           link = "probit",
                           weightit = w.out,
                           br = TRUE)

summary(fit_br)
```

`plot.weightit` *Plot information about the weight estimation process*

Description

`plot.weightit()` plots information about the weights depending on how they were estimated. Currently, only weighting using `method = "gbm"` or `"optweight"` is supported. To plot the distribution of weights, see `plot.summary.weightit()`.

Usage

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

Arguments

`x`	a `weightit` object; the output of a call to `weightit()`.
`...`	unused.

Details

`method = "gbm"`:

After weighting with generalized boosted modeling, `plot()` displays the results of the tuning process used to find the optimal number of trees (and tuning parameter values, if modified) that are used in the final weights. The plot produced has the number of trees on the x-axis and the value of the criterion on the y-axis with a diamond at the optimal point. When multiple parameters are selected by tuning, a separate line is displayed on the plot for each combination of tuning parameters. When by is used in the call to `weightit()`, the plot is faceted by the by variable. See `method_gbm` for more information on selecting tuning parameters.

`method = "optweight"`:

After estimating stable balancing weights, `plot()` displays the values of the dual variables for each balance constraint in a bar graph. Large values of the dual variables indicate the covariates for which the balance constraint is causing increases in the variability of the weights, i.e., the covariates for which relaxing the imbalance tolerance would yield the greatest gains in effective sample size. For continuous treatments, the dual variables are split into those for the target (i.e., ensuring the mean of each covariate after weighting is equal to its unweighted mean) and those for balance (i.e., ensuring the treatment-covariate correlations are no larger than the imbalance tolerance). This is essentially a wrapper for `optweight::plot.optweight()`. See `method_optweight` for details.

Value

A `ggplot` object.

See Also

`weightit()`, `plot.summary.weightit()`

Examples

```
# See example at the corresponding methods page
```

`predict.glm_weightit` *Predictions for* `glm_weightit` *objects*

Description

`predict()` generates predictions for models fit using `glm_weightit()`, `ordinal_weightit()`, `multinom_weightit()`, or `coxph_weightit()`. This page only details the `predict()` methods after using `glm_weightit()`, `ordinal_weightit()`, or `multinom_weightit()`. See `survival::predict.coxph()` for predictions when fitting Cox proportional hazards models using `coxph_weightit()`.

Usage

```
## S3 method for class 'glm_weightit'
predict(object, newdata = NULL, type = "response", na.action = na.pass, ...)

## S3 method for class 'multinom_weightit'
predict(
  object,
  newdata = NULL,
  type = "response",
  na.action = na.pass,
  values = NULL,
  level = NULL,
  ...
)

## S3 method for class 'ordinal_weightit'
predict(
  object,
  newdata = NULL,
  type = "response",
  na.action = na.pass,
  values = NULL,
  level = NULL,
  ...
)
```

Arguments

`object`	a `glm_weightit` object.
`newdata`	optionally, a data frame in which to look for variables with which to predict. If omitted, the fitted values applied to the original dataset are used.

type	the type of prediction desired. Allowable options include "response", for predictions on the scale of the original response variable (also "probs"); "link", for predictions on the scale of the linear predictor (also "lp"); "class", for the modal predicted category for ordinal and multinomial models; "mean", for the expected value of the outcome for ordinal and multinomial models; and "stdlv", for the standardized latent variable values for ordinal models. See Details for more information. The default is "response" for all models, which differs from stats::predict.glm().
na.action	function determining what should be done with missing values in newdata. The default is to predict NA.
...	further arguments passed to or from other methods.
values	when type = "mean", the numeric values each level corresponds to. Should be supplied as a named vector with outcome levels as the names. If NULL and the outcome levels can be converted to numeric, those will be used. See Details.
level	when type = "response" for ordinal and multinomial models, an optional string or number corresponding to the outcome level for which the predictions are to be produced. If NULL (the default), a matrix of predictions for all levels will be produced.

Details

For generalized linear models other than ordinal and multinomial models, see stats::predict.glm() for more information on how predictions are computed and which arguments can be specified. Note that standard errors cannot be computed for the predictions using predict.glm_weightit().

For ordinal and multinomial models, setting type = "mean" computes the expected value of the outcome for each unit; this corresponds to the sum of the values supplied in values weighted by the predicted probability of those values. If values is omitted, predict() will attempt to convert the outcome levels to numeric values, and if this cannot be done, an error will be thrown. values should be specified as a named vector, e.g., values = c(one = 1, two = 2, three = 3), where "one", "two", and "three" are the original outcome levels, and 1, 2, and 3 are the numeric values they correspond to. This method only makes sense to use if the outcome levels meaningfully correspond to numeric values.

For ordinal models, setting type = "link" (also "lp") computes the linear predictor without including the thresholds. This can be interpreted as the prediction of the latent variable underlying the ordinal response. This cannot be used with multinomial models. Setting type = "stdlv" standardizes these predictions by the implied standard deviation of the ordinal responses, which is a function of the link function, the original covariates, and the coefficient estimates.

Value

A numeric vector containing the desired predictions, except for the following circumstances when an ordinal or multinomial model was fit:

- when type = "response" and level = NULL, a numeric matrix with a row for each unit and a column for each level of the outcome with the predicted probability of the corresponding outcome in the cells
- when type = "class", a factor with the modal predicted class for each unit; for ordinal models, this will be an ordered factor.

See Also

stats::predict.glm() for predictions from generalized linear models. glm_weightit() for the fitting function. survival::predict.coxph() for predictions from Cox proportional hazards models.

Examples

```
data("lalonde", package = "cobalt")

# Logistic regression model
fit1 <- glm_weightit(
  re78 > 0 ~ treat * (age + educ + race + married +
                        re74 + re75),
  data = lalonde, family = binomial, vcov = "none")

summary(predict(fit1))

# G-computation using predicted probabilities
p0 <- predict(fit1, type = "response",
              newdata = transform(lalonde,
                                  treat = 0))

p1 <- predict(fit1, type = "response",
              newdata = transform(lalonde,
                                  treat = 1))

mean(p1) - mean(p0)

# Multinomial logistic regression model
lalonde$re78_3 <- factor(findInterval(lalonde$re78,
                                      c(0, 5e3, 1e4)),
                         labels = c("low", "med", "high"))

fit2 <- multinom_weightit(
  re78_3 ~ treat * (age + educ + race + married +
                      re74 + re75),
  data = lalonde, vcov = "none")

# Predicted probabilities
head(predict(fit2))

# Predicted probabilities for a single level
head(predict(fit2, level = "low"))

# Class assignment accuracy
mean(predict(fit2, type = "class") == lalonde$re78_3)

# G-computation using expected value of the outcome
values <- c("low" = 2500,
            "med" = 7500,
            "high" = 12500)
```

```
p0 <- predict(fit2, type = "mean", values = values,
              newdata = transform(lalonde,
                                  treat = 0))

p1 <- predict(fit2, type = "mean", values = values,
              newdata = transform(lalonde,
                                  treat = 1))

mean(p1) - mean(p0)

# Ordinal logistic regression
fit3 <- ordinal_weightit(
  re78 ~ treat * (age + educ + race + married +
                    re74 + re75),
  data = lalonde, vcov = "none")

# G-computation using expected value of the outcome;
# using original outcome values
p0 <- predict(fit3, type = "mean",
              newdata = transform(lalonde,
                                  treat = 0))

p1 <- predict(fit3, type = "mean",
              newdata = transform(lalonde,
                                  treat = 1))

mean(p1) - mean(p0)
```

---

sbps *Subgroup Balancing Propensity Score*

Description

Implements the subgroup balancing propensity score (SBPS), which is an algorithm that attempts to achieve balance in subgroups by sharing information from the overall sample and subgroups (Dong, Zhang, Zeng, & Li, 2020; DZZL). Each subgroup can use either weights estimated using the whole sample, weights estimated using just that subgroup, or a combination of the two. The optimal combination is chosen as that which minimizes an imbalance criterion that includes subgroup as well as overall balance.

Usage

```
sbps(
  obj,
  obj2 = NULL,
  moderator = NULL,
  formula = NULL,
  data = NULL,
```

```
  smooth = FALSE,
  full.search
)
```

Arguments

obj	a weightit object containing weights estimated in the overall sample.
obj2	a weightit object containing weights estimated in the subgroups. Typically this has been estimated by including by in the call to weightit(). Either obj2 or moderator must be specified.
moderator	optional; a string containing the name of the variable in data for which weighting is to be done within subgroups or a one-sided formula with the subgrouping variable on the right-hand side. This argument is analogous to the by argument in weightit(), and in fact it is passed on to by. Either obj2 or moderator must be specified.
formula	an optional formula with the covariates for which balance is to be optimized. If not specified, the formula in obj$call will be used.
data	an optional data set in the form of a data frame that contains the variables in formula or moderator.
smooth	logical; whether the smooth version of the SBPS should be used. This is only compatible with weightit methods that return a propensity score.
full.search	logical; when smooth = FALSE, whether every combination of subgroup and overall weights should be evaluated. If FALSE, a stochastic search as described in DZZL will be used instead. If TRUE, all 2^R combinations will be checked, where R is the number of subgroups, which can take a long time with many subgroups. If unspecified, will default to TRUE if $R \leq 8$ and FALSE otherwise.

Details

The SBPS relies on two sets of weights: one estimated in the overall sample and one estimated within each subgroup. The algorithm decides whether each subgroup should use the weights estimated in the overall sample or those estimated in the subgroup. There are 2^R permutations of overall and subgroup weights, where R is the number of subgroups. The optimal permutation is chosen as that which minimizes a balance criterion as described in DZZL. The balance criterion used here is, for binary and multi-category treatments, the sum of the squared standardized mean differences within subgroups and overall, which are computed using cobalt::col_w_smd(), and for continuous treatments, the sum of the squared correlations between each covariate and treatment within subgroups and overall, which are computed using cobalt::col_w_corr().

The smooth version estimates weights that determine the relative contribution of the overall and subgroup propensity scores to a weighted average propensity score for each subgroup. If P_O are the propensity scores estimated in the overall sample and P_S are the propensity scores estimated in each subgroup, the smooth SBPS finds R coefficients C so that for each subgroup, the ultimate propensity score is $C * P_S + (1 - C) * P_O$, and weights are computed from this propensity score. The coefficients are estimated using optim() with method = "L-BFGS-B". When C is estimated to be 1 or 0 for each subgroup, the smooth SBPS coincides with the standard SBPS.

If obj2 is not specified and moderator is, sbps() will attempt to refit the model specified in obj with the moderator in the by argument. This relies on the environment in which obj was created

to be intact and can take some time if `obj` was hard to fit. It's safer to estimate `obj` and `obj2` (the latter simply by including the moderator in the `by` argument) and supply these to `sbps()`.

Value

A `weightit.sbps` object, which inherits from `weightit`. This contains all the information in `obj` with the weights, propensity scores, call, and possibly covariates updated from `sbps()`. In addition, the `prop.subgroup` component contains the values of the coefficients C for the subgroups (which are either 0 or 1 for the standard SBPS), and the `moderator` component contains a data frame with the moderator.

This object has its own summary method and is compatible with **cobalt** functions. The `cluster` argument should be used with **cobalt** functions to accurately reflect the performance of the weights in balancing the subgroups.

References

Dong, J., Zhang, J. L., Zeng, S., & Li, F. (2020). Subgroup balancing propensity score. *Statistical Methods in Medical Research*, 29(3), 659–676. doi:10.1177/0962280219870836

See Also

`weightit()`, `summary.weightit()`

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups within races
(W1 <- weightit(treat ~ age + educ + married +
                nodegree + race + re74, data = lalonde,
                method = "glm", estimand = "ATT"))

(W2 <- weightit(treat ~ age + educ + married +
                nodegree + race + re74, data = lalonde,
                method = "glm", estimand = "ATT",
                by = "race"))

S <- sbps(W1, W2)

print(S)

summary(S)

bal.tab(S, cluster = "race")

#Could also have run
#S <- sbps(W1, moderator = "race")

S_ <- sbps(W1, W2, smooth = TRUE)
```

```
print(S_)

summary(S_)

bal.tab(S_, cluster = "race")
```

---

summary.weightit *Print and Summarize Output*

Description

summary() generates a summary of the weightit or weightitMSM object to evaluate the properties of the estimated weights. plot() plots the distribution of the weights. nobs() extracts the number of observations.

Usage

```
## S3 method for class 'weightit'
summary(object, top = 5L, ignore.s.weights = FALSE, weight.range = TRUE, ...)

## S3 method for class 'summary.weightit'
plot(x, binwidth = NULL, bins = NULL, ...)

## S3 method for class 'weightitMSM'
summary(
  object,
  top = 5L,
  ignore.s.weights = FALSE,
  weight.range = TRUE,
  which.time,
  ...
)

## S3 method for class 'summary.weightitMSM'
plot(x, binwidth = NULL, bins = NULL, which.time = 1L, ..., time)
```

Arguments

object	a weightit or weightitMSM object; the output of a call to weightit() or weightitMSM().
top	how many of the largest and smallest weights to display. Default is 5. Ignored when weight.range = FALSE.
ignore.s.weights	logical; whether or not to ignore sampling weights when computing the weight summary. If FALSE, the default, the estimated weights will be multiplied by the sampling weights (if any) before values are computed.

`weight.range` `logical`; whether to display statistics about the range of weights and the highest and lowest weights for each group. Default is `TRUE`.

`...` For `plot()`, additional arguments passed to `graphics::hist()` to determine the number of bins, though `ggplot2::geom_histogram()` is actually used to create the plot.

`x` a `summary.weightit` or `summary.weightitMSM` object; the output of a call to `summary.weightit()` or `summary.weightitMSM()`.

`binwidth, bins` arguments passed to `ggplot2::geom_histogram()` to control the size and/or number of bins.

`which.time` for `summary()`, which models to summarize, given as a vector of positions in `formula.list` or of treatment or censoring variable names; omit it to summarize all of them. For `plot()`, which single model to display the distribution of weights for, given as one position or one name; default is the first. This mirrors the argument of the same name in `cobalt::bal.tab()`, except that a value matching no model is an error rather than a warning, since here it decides what is computed. Note that when censoring is modeled, the censoring models occupy positions of their own, so positions do not count treatment time points alone; naming the variable avoids having to count.

`time` for `plot()`, the former name of `which.time`, which still works.

Value

For point treatments (i.e., `weightit` objects), `summary()` returns a `summary.weightit` object with the following elements:

`weight.range` The range (minimum and maximum) weight for each treatment group.

`weight.top` The units with the greatest weights in each treatment group; how many are included is determined by `top`.

`coef.of.var (Coef of Var)`
The coefficient of variation (standard deviation divided by mean) of the weights in each treatment group and overall.

`scaled.mad (MAD)`
The mean absolute deviation of the weights in each treatment group and overall divided by the mean of the weights in the corresponding group.

`negative entropy (Entropy)`
The negative entropy ($\frac{1}{n}\sum w\log(w)$) of the weights in each treatment group and overall, after dividing the weights by their mean in the corresponding group.

`weight.mean (Mean of Weights)`
The mean of the weights in each treatment group and overall. Only included when the weights are stabilized.

`num.zeros` The number of weights equal to zero.

`effective.sample.size`
The effective sample size for each treatment group before and after weighting. See `ESS()`.

For longitudinal treatments (i.e., `weightitMSM` objects), `summary()` returns a list of the above elements for each treatment period. When censoring is modeled (see `.cens()`), each censoring model gets an entry of its own, placed among the treatment entries in the order the models were fit and named for its censoring indicator. Every entry summarizes the same weights – the product across all the models – and differs only in the sample it summarizes them over: a treatment entry splits by treatment group, and a censoring entry covers the units still under observation when that model was fit, since the censored units have a weight of exactly 0 and are not part of the weighted sample.

`plot()` returns a `ggplot` object with a histogram displaying the distribution of the estimated weights. For a censoring model, only the weights of the units still under observation are displayed, matching what `summary()` reports. If the estimand is the ATT or ATC, only the weights for the non-focal group(s) will be displayed (since the weights for the focal group are all 1). A dotted line is displayed at the mean of the weights.

`nobs()` returns a single number. Note that even units with `weights` or `s.weights` of 0 are included.

See Also

`weightit()`, `weightitMSM()`, `summary()`

Examples

```
# See example at ?weightit or ?weightitMSM
```

---

trim *Trim (Winsorize) Large Weights*

Description

Trims (i.e., winsorizes) large weights by setting all weights higher than that at a given quantile to the weight at the quantile or to 0. This can be useful in controlling extreme weights, which can reduce effective sample size by enlarging the variability of the weights. Note that by default, no observations are fully discarded when using `trim()`, which may differ from the some uses of the word "trim" (see the `drop` argument below).

Usage

```
trim(x, ...)

## S3 method for class 'weightit'
trim(x, at = 0, lower = FALSE, drop = FALSE, ...)

## Default S3 method:
trim(x, at = 0, lower = FALSE, treat = NULL, drop = FALSE, ...)
```

Arguments

x	a weightit object or a vector of weights.
...	not used.
at	numeric; either the quantile of the weights above which weights are to be trimmed, given as a single number between .5 and 1 (a value below .5 is replaced by its complement), or the number of weights to be trimmed, given as a number of 1 or greater (e.g., at = 3 for the top 3 weights to be set to the 4th largest weight).
lower	logical; whether also to trim at the lower quantile (e.g., for at = .9, trimming at both .1 and .9, or for at = 3, trimming the top and bottom 3 weights). Default is FALSE to only trim the higher weights.
drop	logical; whether to set the weights of the trimmed units to 0 or not. Default is FALSE to retain all trimmed units. Setting to TRUE may change the original targeted estimand when not the ATT or ATC.
treat	a vector of treatment status for each unit. This should always be included when x is numeric, but you can get away with leaving it out if the treatment is continuous or the estimand is the ATE for binary or multi-category treatments.

Details

trim() takes in a weightit object (the output of a call to weightit() or weightitMSM()) or a numeric vector of weights and trims (winsorizes) them to the specified quantile. All weights above that quantile are set to the weight at that quantile unless drop = TRUE, in which case they are set to 0. If lower = TRUE, all weights below 1 minus the quantile are trimmed. In general, trimming weights can increase imbalance but also decrease the variability of the weights, improving precision at the potential expense of unbiasedness (Cole & Hernán, 2008). See Lee, Lessler, and Stuart (2011) and Thoemmes and Ong (2016) for discussions and simulation results of trimming weights at various quantiles. Note that trimming weights can also change the target population and therefore the estimand.

When using trim() on a numeric vector of weights, it is helpful to include the treatment vector as well. This helps determine the type of treatment and estimand, which are used to specify how trimming is performed. In particular, if the estimand is determined to be the ATT or ATC, the weights of the target (i.e., focal) group are ignored, since they should all be equal to 1. Otherwise, if the estimand is the ATE or the treatment is continuous, all weights are considered for trimming. In general, weights for any group for which all the weights are the same will not be considered in the trimming.

Value

If the input is a weightit object, the output will be a weightit object with the weights replaced by the trimmed weights (or 0) and will have an additional attribute, "trim", equal to the quantile of trimming.

If the input is a numeric vector of weights, the output will be a numeric vector of the trimmed weights, again with the aforementioned attribute.

References

Cole, S. R., & Hernán, M. Á. (2008). Constructing Inverse Probability Weights for Marginal Structural Models. *American Journal of Epidemiology*, 168(6), 656–664. doi:10.1093/aje/kwn164

Lee, B. K., Lessler, J., & Stuart, E. A. (2011). Weight Trimming and Propensity Score Weighting. *PLoS ONE*, 6(3), e18174. doi:10.1371/journal.pone.0018174

Thoemmes, F., & Ong, A. D. (2016). A Primer on Inverse Probability of Treatment Weighting and Marginal Structural Models. *Emerging Adulthood*, 4(1), 40–59. doi:10.1177/2167696815621645

See Also

`weightit()`, `weightitMSM()`

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

(W <- weightit(treat ~ age + educ + married +
                 nodegree + re74, data = lalonde,
               method = "glm", estimand = "ATT"))

summary(W)

#Trimming the top and bottom 5 weights
trim(W, at = 5, lower = TRUE)

#Trimming at 90th percentile
(W.trim <- trim(W, at = .9))

summary(W.trim)
#Note that only the control weights were trimmed

#Trimming a numeric vector of weights
all.equal(trim(W$weights, at = .9, treat = lalonde$treat),
          W.trim$weights)

#Dropping trimmed units
(W.trim <- trim(W, at = .9, drop = TRUE))

summary(W.trim)
#Note that we now have zeros in the control group

#Using made up data and as.weightit()
treat <- rbinom(500, 1, .3)
weights <- rchisq(500, df = 2)
W <- as.weightit(weights, treat = treat,
                 estimand = "ATE")

summary(W)

summary(trim(W, at = .95))
```

weightit *Estimate Balancing Weights*

Description

weightit() allows for the easy generation of balancing weights using a variety of available methods for binary, continuous, and multi-category treatments and for censoring. Some of these methods require functions in other packages, which weightit() calls; these packages must be installed to use the desired method.

Usage

```
weightit(
  formula,
  data = NULL,
  method = "glm",
  estimand = "ATE",
  stabilize = FALSE,
  focal = NULL,
  by = NULL,
  s.weights = NULL,
  ps = NULL,
  missing = NULL,
  verbose = FALSE,
  include.obj = FALSE,
  keep.mparts = TRUE,
  ...
)
```

Arguments

formula	a formula with a treatment variable on the left hand side and the covariates to be balanced on the right hand side. See glm() for more details. Interactions and functions of covariates are allowed. A formula may have an empty right hand side (e.g., A ~ 1), which requests a marginal model at that time point; see *Empty model formulas* in Details. The formulas may also contain **lme4**-style random effects terms (e.g., A ~ X1 + (1 | school)) for methods that accept them, in which case a multilevel model is fit. To estimate censoring weights, use .cens() on the left hand side variable.
data	an optional data set in the form of a data frame that contains the variables in formula.
method	a string of length 1 containing the name of the method that will be used to estimate weights. See Details below for allowable options. The default is "glm" for propensity score weighting using a generalized linear model to estimate the propensity score.

estimand — the desired estimand. For binary and multi-category treatments, can be "ATE", "ATT", "ATC", and, for some methods, "ATO", "ATM", or "ATOS". The default for both is "ATE". This argument is ignored for continuous treatments and censoring weights. See the individual pages for each method for more information on which estimands are allowed with each method and what literature to read to interpret these estimands.

stabilize — whether or not and how to stabilize the weights. If TRUE, each unit's weight will be multiplied by a stabilization factor, which is the unconditional probability (or density) of each unit's observed treatment value. If a formula, a generalized linear model will be fit with the included predictors, and the inverse of the corresponding weight will be used as the stabilization factor. The formula can contain **lme4**-style random effects terms (e.g., `~ (1 | school)`) for methods that accept them, in which case a multilevel model is fit for the numerator. Can only be used when `estimand = "ATE"` or with continuous treatments. Default is FALSE for no stabilization. Note that a continuous treatment's weights already contain the unconditional density as their numerator (see *Continuous Treatments* at method_glm), so `stabilize = TRUE` (equivalently, `~1`) leaves them unchanged.

focal — when `estimand` is set to "ATT" or "ATC", which group to consider the "treated" or "control" group, respectively. This group will not be weighted, and the other groups will be weighted to resemble the focal group. If specified, `estimand` will automatically be set to "ATT" (with a warning if `estimand` is not "ATT" or "ATC"). See section `estimand` *and* `focal` in Details below.

by — a string containing the name of the variable in `data` for which weighting is to be done within categories or a one-sided formula with the stratifying variable on the right-hand side. For example, if `by = "gender"` or `by = ~gender`, a separate propensity score model or optimization will occur within each level of the variable "gender". Only one by variable is allowed; to stratify by multiple variables simultaneously, create a new variable that is a full cross of those variables using `interaction()`.

s.weights — an optional vector of sampling weights or the name of a variable in `data` that contains sampling weights. See the individual pages for each method for information on whether sampling weights can be supplied.

ps — an optional vector of propensity scores or the name of a variable in `data` containing propensity scores. If supplied, `method` is ignored (unless it is a user-supplied function), and the propensity scores will be used to create weights. `formula` must include the treatment variable in `data`, but the listed covariates will play no role in the weight estimation. Using `ps` is similar to calling `get_w_from_ps()` directly, but produces a full `weightit` object rather than just producing weights. See `method_ps` for the accepted formats and the additional arguments available.

missing — `character`; how missing data should be handled. The options and defaults depend on the `method` used. Ignored if no missing data is present. It should be noted that multiple imputation outperforms all available missingness methods available in `weightit()` and should probably be used instead. Consider the **MatchThem** package for the use of `weightit()` with multiply imputed data.

verbose — `logical`; whether to print additional information output by the fitting function. Default is FALSE to suppress output.

include.obj — logical; whether to include in the output any fit objects created in the process of estimating the weights. For example, with method = "glm", the glm objects containing the propensity score model will be included. See the individual pages for each method for information on what object will be included if TRUE. Default is FALSE to keep the returned object small.

keep.mparts — logical; whether to include in the output components necessary to estimate standard errors that account for estimation of the weights in glm_weightit(). Default is TRUE if such parts are present. See the individual pages for each method for whether these components are produced. Set to FALSE to keep the output object smaller, e.g., if standard errors will not be computed using glm_weightit().

... — other arguments for functions called by weightit() that control aspects of fitting that are not covered by the above arguments. See Details.

Details

The primary purpose of weightit() is as a dispatcher to functions that perform the estimation of balancing weights using the requested method. Below are the methods allowed and links to pages containing more information about them, including additional arguments and outputs (e.g., when include.obj = TRUE), how missing values are treated, which estimands are allowed, and whether sampling weights are allowed.

method	Name
"glm"	Propensity score weighting using generalized linear models
"gbm"	Propensity score weighting using generalized boosted modeling
"cbps"	Covariate Balancing Propensity Score weighting
"npcbps"	Nonparametric Covariate Balancing Propensity Score weighting
"ebal"	Entropy balancing
"ipt"	Inverse probability tilting
"optweight"	Stable balancing weights
"super"	Propensity score weighting using SuperLearner
"bart"	Propensity score weighting using Bayesian additive regression trees (BART)
"energy"	Energy balancing
"cfd"	Characteristic function distance balancing

method can also be supplied as a user-defined function; see method_user for instructions and examples. Setting method = NULL computes unit weights.

Empty model formulas:

The right hand side of formula may be empty, as in A ~ 1, requesting a marginal model in which the treatment (or censoring) is taken to be independent of the covariates. With no covariates there is nothing for any method to model or balance, and every method's target is met by the same weights: the inverse of the marginal treatment probability, or $1/P(C = 0)$ for a censoring model. Those weights are computed by fitting an intercept-only generalized linear model whatever method is supplied, which is simply the easiest way to get them.

For a continuous treatment the conditional density of the treatment is its marginal density, so all the weights are exactly 1 and nothing is estimated; no M-estimation components are produced.

For the other treatment types the marginal probability is estimated, so M-estimation is available as usual.

A formula whose only terms are **lme4**-style random effects, such as `A ~ (1 | school)`, is *not* empty in this sense and is fit as the multilevel model it describes.

Censoring weights (IPCW):

Wrapping the left side of `formula` in `.cens()` requests inverse probability of censoring weights instead of treatment weights, as in `weightit(.cens(C) ~ x1 + x2, data = d, method = "glm")`. Censoring is treated as its own treatment type, distinct from binary, multi-category, and continuous treatments; the indicator must be 0 or `FALSE` for units still under observation and 1 or `TRUE` for units that are censored.

Weights are estimated only for the units still under observation, and are those that make their covariate distribution resemble that of the full at-risk sample. Writing $e(X) = P(C = 1|X)$, the weights are $1/(1 - e(X))$ for units with $C = 0$ and exactly 0 for units with $C = 1$.

`estimand`, `focal`, and `subclass` do not apply and are rejected or ignored. `by` and `stabilize` can be used; stabilization multiplies the weights by $P(C = 0|V)$ from a second censoring model (marginal when `stabilize` is `TRUE`, otherwise fit with the predictors in the supplied formula), giving $P(C = 0|V)/P(C = 0|X)$ for the units still under observation and leaving the censored units at exactly 0. `ps` is the predicted probability of *being censored* (i.e., of having missing outcomes). Not all methods support estimating censoring weights; see the `treat_type` component of `.weightit_methods` and each method's help page.

As with a treatment model, the right side of the formula may be empty, as in `.cens(C) ~ 1`, which requests a marginal censoring model that assumes censoring is independent of the covariates. The resulting weights are $1/P(C = 0)$ for the units still under observation and 0 for the censored units. The rest of the censoring machinery is unaffected, so such a model can be combined with `by`, `s.weights`, `stabilize` (which then makes all nonzero weights exactly 1), and M-estimation, and it can be interleaved with covariate-dependent censoring models in `weightitMSM()`. See *Empty model formulas* above for how `method` is handled.

Because censored units receive a weight of exactly 0, they contribute nothing to a weighted outcome model, and `glm_weightit()` and friends tolerate missing values in the model variables for those units, including a missing event time in the `Surv()` response of a `coxph_weightit()` model. Missing values in units with a nonzero weight still produce an error.

`estimand` **and** `focal`**:**

For binary and multi-category treatments, the argument to `estimand` determines what distribution the weighted sample should resemble. When set to `"ATE"`, this requests that each group resemble the full sample. When set to `"ATT"` or `"ATC"`, this requests that each group resemble the treated or control group, respectively (termed the "focal" group); weights are set to 1 for the focal group. When set to `"ATO"`, `"ATM"`, or `"ATOS"` (for the methods that allow them), this requests that each group resemble an "overlap" sample.

How does `weightit()` decide which group is the treated and which group is the control? For binary treatments, several heuristics are used. The first is by checking whether a valid argument to `focal` was supplied containing the name of the focal group, which is the treated group when `estimand = "ATT"` and the control group when `estimand = "ATC"`. If `focal` is not supplied, guesses are made using the following criteria, evaluated in order:

- If the treatment variable is `logical`, `TRUE` is considered treated and `FALSE` control.

- If the treatment is numeric (or a string or factor with values that can be coerced to numeric values), if 0 is one of the values, it is considered the control, and otherwise, the lower value is considered the control (with the other considered treated).
- If exactly one of the treatment values is `"t"`, `"tr"`, `"treat"`, `"treated"`, or `"exposed"`, it is considered the treated (and the other control).
- If exactly one of the treatment values is `"c"`, `"co"`, `"ctrl"`, `"control"`, or `"unexposed"`, it is considered the control (and the other treated).
- If the treatment variable is a factor, the first level is considered control and the second treated.
- The lowest value after sorting with `sort()` is considered control and the other treated.

To be safe, it is best to code your binary treatment variable as `0` for control and `1` for treated. Otherwise, `focal` should be supplied when requesting the ATT or ATC. For multi-category treatments, `focal` is required when requesting the ATT or ATC; none of the heuristics above are used.

Citing WeightIt:

When using `weightit()`, please cite both the **WeightIt** package (using `citation("WeightIt")`) and the paper(s) in the references section of the method used.

Value

A `weightit` object with the following elements:

`weights`	The estimated weights, one for each unit.
`treat`	The values of the treatment variable.
`covs`	The covariates used in the fitting. Only includes the raw covariates, which may have been altered in the fitting process.
`estimand`	The estimand requested.
`method`	The weight estimation method specified.
`ps`	The estimated or provided propensity scores. Estimated propensity scores are returned for binary treatments and only when `method` is one that estimates propensity scores. The propensity score corresponds to the predicted probability of being treated; see section `estimand` *and* `focal` in Details for how the treated group is determined.
`s.weights`	The provided sampling weights, or a vector of 1s if none are provided.
`focal`	The focal treatment level if the ATT or ATC was requested.
`by`	A data frame containing the by variable when specified.
`obj`	When `include.obj = TRUE`, the fit object.
`info`	Additional information about the fitting. See the individual methods pages for what is included.

When `keep.mparts` is `TRUE` (the default) and the chosen method is compatible with M-estimation, the components related to M-estimation for use in `glm_weightit()` are stored in the `"Mparts"` attribute. When by is specified, the per-stratum M-estimation components are instead combined and stored in the `"Mparts.list"` attribute; the resulting standard errors produced by `glm_weightit()` are asymptotically equivalent to those from estimating the weights from a single model in which the by variable is fully interacted with all the covariates. The same is true for `weightitMSM()`, where the equivalent model interacts the by variable with all the covariates at every time point.

See Also

`weightitMSM()` for estimating weights with sequential (i.e., longitudinal) treatments or with both treatment and censoring indicators for use in estimating marginal structural models (MSMs).

`weightit.fit()`, which is a lower-level dispatcher function that accepts a matrix of covariates and a vector of treatment statuses rather than a formula and data frame and performs minimal argument checking and processing. It may be useful for speeding up simulation studies for which the correct arguments are known. In general, `weightit()` should be used.

`summary.weightit()` for summarizing the distribution of the weights.

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

#Balancing covariates between treatment groups (binary)
(W1 <- weightit(treat ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "glm", estimand = "ATT"))
summary(W1)
bal.tab(W1)

#Balancing covariates with respect to race (multi-category)
(W2 <- weightit(race ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "ebal", estimand = "ATE"))
summary(W2)
bal.tab(W2)

#Balancing covariates with respect to re75 (continuous)
(W3 <- weightit(re75 ~ age + educ + married +
                  nodegree + re74, data = lalonde,
                method = "cbps"))
summary(W3)
bal.tab(W3)
```

---

weightit.fit *Generate Balancing Weights with Minimal Input Processing*

Description

`weightit.fit()` dispatches one of the weight estimation methods determined by `method`. It is an internal function called by `weightit()` and should probably not be used except in special cases. Unlike `weightit()`, `weightit.fit()` does not accept a formula and data frame interface and instead requires the covariates and treatment to be supplied as a numeric matrix and atomic vector, respectively. In this way, `weightit.fit()` is to `weightit()` what `lm.fit()` is to `lm()`: a thinner, slightly faster interface that performs minimal argument checking.

Usage

```
weightit.fit(
  covs,
  treat,
  method = "glm",
  s.weights = NULL,
  by.factor = NULL,
  estimand = "ATE",
  focal = NULL,
  stabilize = FALSE,
  ps = NULL,
  missing = NULL,
  verbose = FALSE,
  include.obj = FALSE,
  subset = NULL,
  ...
)
```

Arguments

covs	a numeric matrix of covariates. May have zero columns, in which case the weights are computed with an intercept-only generalized linear model whatever method is supplied and any method-specific arguments are ignored; see *Empty model formulas* in Details at weightit().
treat	a vector of treatment statuses. To request inverse probability of censoring weights, supply a 0/1 censoring indicator (1 = censored) tagged with .cens(), as in treat = .cens(C); NA values are then allowed outside subset, since a unit censored at an earlier time point has no later indicator.
method	a string containing the name of the method that will be used to estimate weights. See weightit() for allowable options. The default is "glm" for propensity score weighting using a generalized linear model to estimate the propensity score.
s.weights	a numeric vector of sampling weights. See the individual pages for each method for information on whether sampling weights can be supplied.
by.factor	a factor variable for which weighting is to be done within levels. Corresponds to the by argument in weightit().
estimand	the desired estimand. For binary and multi-category treatments, can be "ATE", "ATT", "ATC", and, for some methods, "ATO", "ATM", or "ATOS". The default for both is "ATE". This argument is ignored for continuous treatments and censoring weights. See the individual pages for each method for more information on which estimands are allowed with each method and what literature to read to interpret these estimands.
focal	when estimand is set to "ATT" or "ATC", which group to consider the "treated" or "control" group. This group will not be weighted, and the other groups will be weighted to resemble the focal group. If specified, estimand will automatically be set to "ATT" (with a warning if estimand is not "ATT" or "ATC"). See section estimand *and* focal in Details at weightit().

stabilize	logical; whether or not to stabilize the weights. For the methods that involve estimating propensity scores, this involves multiplying each unit's weight by the proportion of units in their treatment group. Default is FALSE. Note this differs from its use with weightit().
ps	a vector of propensity scores. If specified, method will be ignored and set to "glm".
missing	character; how missing data should be handled. The options depend on the method used. If NULL, covs will be checked for NA values, and if present, missing will be set to "ind". If "", covs will not be checked for NA values; this can be faster when it is known there are none.
verbose	logical; whether to print additional information output by the fitting function. Default is FALSE to suppress output.
include.obj	logical; whether to include in the output any fit objects created in the process of estimating the weights. For example, with method = "glm", the glm objects containing the propensity score model will be included. See the individual pages for each method for information on what object will be included if TRUE. Default is FALSE to keep the returned object small.
subset	a logical vector with length equal to that of treat restricting which units are used to fit the model, e.g., the units still under observation when a censoring model is fit. The weights of the remaining units are left as NA for the caller to fill in. When by.factor is also supplied, the model is fit within the intersection of each by level and subset. If NULL (the default), all units are used.
...	other arguments for functions called by weightit.fit() that control aspects of fitting that are not covered by the above arguments.

Details

weightit.fit() is called by weightit() after the arguments to weightit() have been checked and processed. weightit.fit() dispatches the function used to actually estimate the weights, passing on the supplied arguments directly. weightit.fit() is not meant to be used by anyone other than experienced users who have a specific use case in mind. The returned object contains limited information about the supplied arguments or details of the estimation method; all that is processed by weightit().

Less argument checking or processing occurs in weightit.fit() than does in weightit(), which means supplying incorrect arguments can result in errors, crashes, and invalid weights, and error and warning messages may not be helpful in diagnosing the problem. weightit.fit() does check to make sure weights were actually estimated, though.

weightit.fit() may be most useful in speeding up simulation studies that use weightit() because the covariates can be supplied as a numeric matrix, which is often how they are generated in simulations, without having to go through the potentially slow process of extracting the covariates and treatment from a formula and data frame. If the user is certain the arguments are valid (e.g., by ensuring the estimated weights are consistent with those estimated from weightit() with the same arguments), less time needs to be spent on processing the arguments. Also, the returned object is much smaller than a weightit object because the covariates are not returned alongside the weights.

Value

A `weightit.fit` object with the following elements:

`weights`	The estimated weights, one for each unit.
`treat`	The values of the treatment variable.
`estimand`	The estimand requested.
`method`	The weight estimation method specified.
`ps`	The estimated or provided propensity scores. Estimated propensity scores are returned for binary treatments and censoring models, and only when `method` is `"glm"`, `"gbm"`, `"cbps"`, `"ipt"`, `"super"`, or `"bart"`. For binary treatments, the propensity score corresponds to the predicted probability of being treated; see section `estimand` *and* `focal` in Details at `weightit()` for how the treated group is determined. For censoring models, it corresponds to the predicted probability of *being censored*, i.e., of the censoring indicator equaling 1.
`s.weights`	The provided sampling weights.
`focal`	The focal treatment level if the ATT or ATC was requested.
`fit.obj`	When `include.obj = TRUE`, the fit object.
`info`	Additional information about the fitting. See the individual methods pages for what is included.

The `weightit.fit` object does not have specialized `print()`, `summary()`, or `plot()` methods. It is simply a list containing the above components. Use `as.weightit()` to convert it to a `weightit` object, which does have these methods. See Examples.

See Also

`weightit()`, which you should use for estimating weights unless you know better.

`as.weightit()` for converting a `weightit.fit` object to a `weightit` object.

Examples

```
library("cobalt")
data("lalonde", package = "cobalt")

# Balancing covariates between treatment groups (binary)
covs <- lalonde[c("age", "educ", "race", "married",
                  "nodegree", "re74", "re75")]
## Create covs matrix, splitting any factors using
## cobalt::splitfactor()
covs_mat <- as.matrix(splitfactor(covs))

WF1 <- weightit.fit(covs_mat, treat = lalonde$treat,
                    method = "glm", estimand = "ATT")
str(WF1)

# Converting to a weightit object for use with
# summary() and bal.tab()
```

```
W1 <- as.weightit(WF1, covs = covs)
W1
summary(W1)
bal.tab(W1)
```

---

weightitMSM *Generate Balancing Weights for Longitudinal Treatments*

Description

`weightitMSM()` allows for the easy generation of balancing weights for marginal structural models for time-varying treatments using a variety of available methods for binary, continuous, and multi-category treatments, as well as censoring. Some of these methods exist in other packages, which `weightit()` calls; these packages must be installed to use the desired method.

Usage

```
weightitMSM(
  formula.list,
  data = NULL,
  method = "glm",
  stabilize = FALSE,
  by = NULL,
  s.weights = NULL,
  num.formula = NULL,
  missing = NULL,
  verbose = FALSE,
  include.obj = FALSE,
  keep.mparts = TRUE,
  is.MSM.method,
  weightit.force = FALSE,
  ...
)
```

Arguments

`formula.list` a list of formulas corresponding to each time point with the time-specific treatment variable on the left hand side and pre-treatment covariates to be balanced on the right hand side. The formulas must be in temporal order, and must contain all covariates to be balanced at that time point (i.e., treatments and covariates featured in early formulas should appear in later ones). Interactions and functions of covariates are allowed. As in `weightit()`, a formula may have an empty right hand side (e.g., `A_1 ~ 1`), which requests a marginal model at that time point; see *Empty model formulas* in Details at `weightit()`. The formulas may also contain **lme4**-style random effects terms (e.g., `A_1 ~ X1_0 + (1 | school)`) for methods that accept them, in which case a multilevel model is fit at that time point. Note this is intended for clustering, not for modeling the longitudinal treatments in a single model.

`data` an optional data set in the form of a data frame that contains the variables in the formulas in `formula.list`. This must be a wide data set with exactly one row per unit.

`method` a string of length 1 containing the name of the method that will be used to estimate weights. See `weightit()` for allowable options. The default is `"glm"`, which estimates the weights using generalized linear models.

`stabilize` whether or not and how to stabilize the weights. Stabilizing the weights involves fitting a model predicting treatment at each time point from treatment status at prior time points. If `TRUE`, a fully saturated model will be fit (i.e., all interactions between all treatments up to each time point), essentially using the observed treatment probabilities in the numerator (for binary and multi-category treatments). This may yield an error if some combinations are not observed, and with many time points, saturated models may be time-consuming or impossible to fit. Can also be a one-sided formula or list thereof, in which case `stabilize` replaces `num.formula` (described below); as with `formula.list`, these formulas may contain **lme4**-style random effects terms. Default is `FALSE` for unstabilized weights.

`by` a string containing the name of the variable in `data` for which weighting is to be done within categories or a one-sided formula with the stratifying variable on the right-hand side. For example, if `by = "gender"` or `by = ~gender`, a separate propensity score model or optimization will occur within each level of the variable `"gender"`. Only one by variable is allowed; to stratify by multiple variables simultaneously, create a new variable that is a full cross of those variables using `interaction()`.

`s.weights` an optional vector of sampling weights or the name of a variable in `data` that contains sampling weights. See the individual pages for each method for information on whether sampling weights can be supplied.

`num.formula` an optional one-sided formula with the stabilization factors (other than the previous treatments) on the right hand side, which adds, for each time point, the stabilization factors to a model saturated with previous treatments. See Cole & Hernán (2008) for a discussion of how to specify this model; including stabilization factors can change the estimand without proper adjustment, and should be done with caution. Can also be a list of one-sided formulas, one for each entry of `formula.list`, including any censoring entries. Ignored if `stabilize` is a formula or list thereof.

`missing` `character`; how missing data should be handled. The options and defaults depend on the `method` used. Ignored if no missing data is present. It should be noted that multiple imputation outperforms all available missingness methods available in `weightit()` and should probably be used instead. Consider the **MatchThem** package for the use of `weightit()` with multiply imputed data.

`verbose` `logical`; whether to print additional information output by the fitting function. Default is `FALSE` to suppress output.

`include.obj` `logical`; whether to include in the output a list of the fit objects created in the process of estimating the weights at each time point. For example, with `method = "glm"`, a list of the `glm` objects containing the propensity score models at each time point will be included. See the help pages for each method for information on what object will be included if `TRUE`.

keep.mparts	logical; whether to include in the output components necessary to estimate standard errors that account for estimation of the weights in glm_weightit(). Default is TRUE if such parts are present. See the individual pages for each method for whether these components are produced. Set to FALSE to keep the output object smaller, e.g., if standard errors will not be computed using glm_weightit().
is.MSM.method	logical; whether the method estimates weights for multiple time points all at once (TRUE) or by estimating weights at each time point and then multiplying them together (FALSE). This is only relevant for user-specified functions.
weightit.force	logical; several methods are not valid for estimating weights with longitudinal treatments, and will produce an error message if attempted. Set to TRUE to bypass this error message.
...	other arguments that control aspects of fitting that are not covered by the above arguments. See Details at weightit().

Details

In general, weightitMSM() works by separating the estimation of weights into separate procedures for each time period based on the formulas provided. For each formula, weightitMSM() simply applies weightit() to that formula, collects the weights for each time period, and multiplies them together to arrive at longitudinal balancing weights.

Each formula should contain all the covariates to be balanced on. For example, the formula corresponding to the second time period should contain all the baseline covariates, the treatment variable at the first time period, and the time-varying covariates that took on values after the first treatment and before the second. Currently only "wide" data sets are supported, where each unit is represented by exactly one row that contains its covariate and treatment history encoded in separate variables. You can use reshape() or other functions to transform a long data set into this format.

Censoring weights (IPCW):

Censoring can be modeled by including entries in formula.list whose left side is wrapped in .cens(), placed in temporal order among the treatment models. For example,

```
weightitMSM(list(A_1 ~ X1_0 + X2_0,
                 A_2 ~ X1_1 + X2_1 + A_1,
                 .cens(C_2) ~ X1_1 + X2_1 + A_1 + A_2,
                 A_3 ~ X1_2 + X2_2 + A_2),
            data = d, method = "glm")
```

models censoring occurring after the second treatment. Each censoring indicator must be 0 for units still under observation and 1 for units censored at that time point. See .cens() for details of what the resulting weights estimate.

Every model, treatment or censoring, is fit only among the units still under observation when it is reached, and the resulting weights are multiplied together across time points as usual. A unit censored at any time point therefore has a final weight of exactly 0. Because such units drop out, missing values are permitted in the treatments and covariates that follow their censoring; missing values among units still under observation remain an error. at.risk in the output has one column per time point recording which units were under observation when that model was fit.

Censoring time points are stabilized in exactly the same way as treatment time points: with `stabilize = TRUE`, the numerator of a censoring weight is a model for that censoring indicator given the preceding treatments (or a marginal model when no treatment precedes it), and `num.formula` adds stabilization factors to it as it does for a treatment. When `num.formula` is supplied as a list, it must have one entry per entry of `formula.list`, censoring entries included. The numerator of a censoring weight is itself a censoring model, so the stabilized weight is $P(C = 0|\cdot)/P(C = 0|X)$ for the units still under observation and remains exactly 0 for those censored.

The right side of a censoring formula may be empty, as in `.cens(C_2) ~ 1`, which requests a marginal censoring model that assumes censoring at that time point is independent of the covariates; its contribution to the product is $1/P(C = 0)$ for the units still under observation and 0 for those censored there. Everything else is unaffected: the risk sets, the missing values permitted after censoring, `stabilize`, `by`, and M-estimation all work as they do for a covariate-dependent censoring model, and empty and non-empty censoring formulas can be mixed freely. Each time point is fit separately, so only the empty ones take the intercept-only shortcut described in *Empty model formulas* in Details at `weightit()`; when `is.MSM.method = TRUE` there is no shortcut to take, because a single set of weights is estimated for all time points at once, and a time point with no covariates instead contributes only its intercept balance condition.

Value

A `weightitMSM` object with the following elements:

`weights`	The estimated weights, one for each unit.
`treat.list`	A list of the values of the time-varying treatment variables, one entry per entry of `formula.list` and named for the variable modeled there. When censoring is modeled, the censoring indicators are included in place among the treatments, in the order they were fit; an entry's `"treat.type"` attribute is `"censoring"` when it is a censoring indicator, which is 0 for units still under observation, 1 for units censored at that time point, and `NA` for units censored earlier.
`covs.list`	A list of the covariates used in the fitting at each time point, aligned with `treat.list`. Only includes the raw covariates, which may have been altered in the fitting process.
`estimand`	"ATE", currently the only estimand for MSMs with binary or multi-category treatments.
`method`	The weight estimation method specified.
`s.weights`	The provided sampling weights.
`by`	A data frame containing the by variable when specified.
`stabilization`	The stabilization factors, if any.

When censoring is modeled (i.e., when any entry of `formula.list` has its left side wrapped in `.cens()`), `formula.list` is kept exactly as supplied, markers included, so that `update()` round-trips, and one additional component is returned:

`at.risk`	A logical matrix with one row per unit and one column per entry of `formula.list`, named for the treatment or censoring variable modeled at that entry. Each column records which units were still under observation when that model was fit, i.e., which units contributed to it. Useful for assessing balance; see `.cens()`.

Censored units have a final weight of exactly 0, so `weights` is 0 for any unit censored at any time point.

When `keep.mparts` is `TRUE` (the default) and the chosen method is compatible with M-estimation, the components related to M-estimation for use in `glm_weightit()` are stored in the `"Mparts.list"` attribute. When by is specified, the per-stratum components are combined into `"Mparts.list"` so that the standard errors produced by `glm_weightit()` are asymptotically equivalent to those from estimating the weights from models in which the by variable is fully interacted with all the covariates at every time point. (For `method = "cbps"`, this requires `is.MSM.method = FALSE`, i.e., estimating a separate model at each time point, as M-estimation is not supported for the single-model MSM version of CBPS.)

References

Cole, S. R., & Hernán, M. A. (2008). Constructing Inverse Probability Weights for Marginal Structural Models. *American Journal of Epidemiology*, 168(6), 656–664. doi:10.1093/aje/kwn164

See Also

`weightit()` for information on the allowable methods

`summary.weightitMSM()` for summarizing the weights

Examples

```
data("msmdata")
(W1 <- weightitMSM(list(A_1 ~ X1_0 + X2_0,
                        A_2 ~ X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0,
                        A_3 ~ X1_2 + X2_2 +
                          A_2 + X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0),
                   data = msmdata,
                   method = "glm"))
summary(W1)
cobalt::bal.tab(W1)

# Using stabilization factors
W2 <- weightitMSM(list(A_1 ~ X1_0 + X2_0,
                        A_2 ~ X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0,
                        A_3 ~ X1_2 + X2_2 +
                          A_2 + X1_1 + X2_1 +
                          A_1 + X1_0 + X2_0),
                   data = msmdata,
                   method = "glm",
                   stabilize = TRUE,
                   num.formula = list(~ 1,
                                      ~ A_1,
                                      ~ A_1 + A_2))

# Same as above but with fully saturated stabilization factors
# (i.e., making the last entry in 'num.formula' A_1*A_2)
```

```
W3 <- weightitMSM(list(A_1 ~ X1_0 + X2_0,
                       A_2 ~ X1_1 + X2_1 +
                         A_1 + X1_0 + X2_0,
                       A_3 ~ X1_2 + X2_2 +
                         A_2 + X1_1 + X2_1 +
                         A_1 + X1_0 + X2_0),
                  data = msmdata,
                  method = "glm",
                  stabilize = TRUE)
```

Index

∗ **datasets**
    msmdata, 86
.cens, 3
.cens(), *6*, *14*, *17, 18*, *30*, *34*, *39*, *43*, *48*, *53*, *60*, *66*, *72*, *76*, *78*, *104*, *107*, *110*, *113*, *118, 119*
.weightit_methods, *3*, *5*, 5, *110*

anova.glm(), *9*
anova.glm_weightit, 8
anova.glm_weightit(), *25*, *28*
as.weightit, 9
as.weightit(), *115*
as.weightitMSM *(*as.weightit*)*, 9

binomial(), *61, 62*
brglm2::bracl(), *93*
brglm2::brglmControl(), *23*
brglm2::brglmFit(), *23*
brglm2::brmultinom(), *89, 90*

calibrate, 11
CBPS::npCBPS(), *69, 70*
cobalt::.cens(), *3*, *5*
cobalt::bal.compute(), *54*, *80*
cobalt::bal.tab(), *3, 4*, *103*
cobalt::col_w_corr(), *100*
cobalt::col_w_smd(), *100*
cobalt::treat, *4*
coef, *26*
confint(), *28*
confint.lm(), *27*
coxph_weightit, 13
coxph_weightit(), *3*, *24*, *90*, *94*, *110*

dbarts::bart2(), *29–31*
density(), *30*, *32*, *53*, *56*, *60*, *63*, *76*, *78*, *80*
dist(), *49, 50*
dnorm(), *32*, *56*, *62*, *76*, *79*

entropy balancing, *70*

ESS, 15
ESS(), *103*
estfun.glm_weightit *(*glm_weightit-methods*)*, 25

family, *22*
fitted(), *32*
format, *26*
formula, *13*, *22*, *87*, *91*
fwb::fwb(), *14*, *22, 23*, *88*, *92*

gaussian(), *63*
gbm::gbm.fit(), *53*, *55–57*
gbm::gbmCrossVal(), *55*
get_w_from_ps, 17
get_w_from_ps(), *3*, *6*, *29*, *32, 33*, *53*, *55*, *59, 60*, *62*, *65*, *75–79*, *81*, *108*
ggplot2::geom_histogram(), *103*
glm(), *23, 24*, *59*, *61–63*, *107*
glm.control(), *63*
glm.fit(), *61*
glm_weightit, 21
glm_weightit(), *3*, *9*, *15*, *23*, *25–28*, *35*, *44*, *60, 61*, *67*, *75*, *89, 90*, *93, 94*, *98*, *109–111*, *118*, *120*
glm_weightit-methods, 25
graphics::hist(), *103*

interaction(), *108*, *117*

lm(), *24*, *112*
lm.fit(), *112*
lm_weightit *(*glm_weightit*)*, 21
lm_weightit(), *3*
lme4::glmer(), *30*, *60*, *63*
lme4::lmer(), *60*, *63*
logistf::logistf.control(), *61*

make_full_rank, 28
make_full_rank(), *83*
MASS::polr(), *62*, *94*

match.fun(), *26*
mclogit::mblogit(), *60*, *62, 63*, *90*
method_bart, 29
method_cbps, 33, *46*, *68*, *70*
method_cfd, 39
method_ebal, *38*, 43, *68*, *74*
method_energy, 47
method_entropy *(*method_ebal*)*, 43
method_gbm, 52, *95*
method_glm, *20*, *33*, 59, *76, 77*, *81*, *108*
method_ipt, *38*, *46*, 66
method_kernel *(*method_cfd*)*, 39
method_npcbps, 69, *74*
method_optweight, *46*, *70*, 71, *95*
method_ps, 75, *108*
method_sbw *(*method_optweight*)*, 71
method_super, *33*, 77
method_user, *28, 29*, 82, *109*
misaem::miss.glm(), *63*
misaem::miss.lm(), *63*
misaem::predict.miss.glm(), *63*
misaem::predict.miss.lm(), *63*
MNP::MNP(), *62*
model.frame(), *24*
model.matrix(), *23*, *29*, *88*, *92*
msmdata, 86
multinom_weightit, 87
multinom_weightit(), *3*, *15*, *24*, *62*, *64*, *94*

offset(), *23*, *88*, *92*
optim(), *34*, *43*, *100*
optweight::optweight(), *71*, *74*
optweight::optweight.fit(), *71–74*
optweight::optweight.svy.fit(), *72*
optweight::plot.optweight(), *95*
ordinal_weightit, 90
ordinal_weightit(), *3*, *15*, *24*, *62*, *64*, *90*
osqp::osqp(), *39*, *47*
osqp::osqpSettings(), *40, 41*, *49*, *51*
osqp::solve_osqp(), *42*, *51*

plot.summary.weightit
*(*summary.weightit*)*, 102
plot.summary.weightit(), *95*
plot.summary.weightitMSM
*(*summary.weightit*)*, 102
plot.weightit, 95
plot.weightit(), *54*, *73*
predict.glm_weightit, 96
predict.glm_weightit(), *25*, *28*
predict.multinom_weightit
*(*predict.glm_weightit*)*, 96
predict.ordinal_weightit
*(*predict.glm_weightit*)*, 96
print, *26*
print.glm_weightit
*(*glm_weightit-methods*)*, 25

qr(), *28*
quasibinomial(), *36*, *67*

reshape(), *118*
rootSolve::multiroot(), *36, 37*, *45*, *66*, *68*

sandwich::bread(), *28*
sandwich::estfun(), *28*
sandwich::sandwich(), *27*
sbps, 99
set.seed(), *31*, *55*
sort(), *111*
stan4bart::stan4bart(), *30–32*
stats::optim(), *36*, *45*
stats::predict.glm(), *97, 98*
summary(), *104*
summary.coxph_weightit
*(*glm_weightit-methods*)*, 25
summary.glm(), *28*
summary.glm_weightit
*(*glm_weightit-methods*)*, 25
summary.multinom_weightit
*(*glm_weightit-methods*)*, 25
summary.ordinal_weightit
*(*glm_weightit-methods*)*, 25
summary.weightit, 102
summary.weightit(), *16*, *101*, *112*
summary.weightitMSM *(*summary.weightit*)*,
102
summary.weightitMSM(), *120*
SuperLearner::listWrappers(), *79*
SuperLearner::SuperLearner(), *78–80*
Surv(), *13*
survival::coxph(), *13–15*
survival::coxph.control(), *14*
survival::predict.coxph(), *96*, *98*

trim, 104
trim(), *55*

update(), *119*

update.formula(), *27*
update.glm_weightit
(glm_weightit-methods), 25
user-defined methods, *28*

vcov(), *28*
vcov.glm_weightit
(glm_weightit-methods), 25

weightit, 107
weightit(), *3–7*, *10*, *12*, *13*, *22*, *28*, *29*, *31*, *33*, *34*, *38*, *39*, *42*, *43*, *46*, *47*, *51*, *52*, *57*, *59*, *65*, *66*, *68–71*, *74*, *75*, *77*, *81*, *83*, *84*, *87*, *91*, *95*, *100–102*, *104–106*, *112–120*
weightit.fit, 112
weightit.fit(), *3*, *6*, *10*, *112*
weightitMSM, 116
weightitMSM(), *3–7*, *10*, *12*, *13*, *22*, *29*, *33*, *34*, *38*, *39*, *42–44*, *46–48*, *51*, *52*, *57*, *59*, *65*, *66*, *68–70*, *72*, *74*, *75*, *77*, *81*, *83*, *84*, *87*, *91*, *102*, *104–106*, *110–112*