Package 'MatchIt'

July 21, 2025

Version 4.7.2

Title Nonparametric Preprocessing for Parametric Causal Inference

Description Selects matched samples of the original treated and control groups with similar covariate distributions -- can be used to match exactly on covariates, to match on propensity scores, or perform a variety of other matching procedures. The package also implements a series of recommendations offered in Ho, Imai, King, and Stuart (2007) <DOI:10.1093/pan/mpl013>. (The 'gurobi' package, which is not on CRAN, is optional and comes with an installation of the Gurobi Optimizer, available at <https://www.gurobi.com>.)

Depends R (>= 3.6.0)

Imports backports (>= 1.1.9), chk (>= 0.10.0), rlang (>= 1.1.0), Rcpp, utils, stats, graphics, grDevices

Suggests optmatch (>= 0.10.6), Matching, rgenoud, quickmatch (>= 0.2.1), nnet, rpart, mgcv, CBPS (>= 0.17), dbarts (>= 0.9-28), randomForest (>= 4.7-1), glmnet (>= 4.0), gbm (>= 2.1.7), cobalt (>= 4.2.3), boot, marginaleffects (>= 0.25.0), sandwich (>= 2.5-1), survival, RcppProgress (>= 0.4.2), highs, Rglpk, Rsymphony, gurobi, knitr, rmarkdown, testthat (>= 3.0.0)

LinkingTo Rcpp, RcppProgress

Encoding UTF-8

LazyData true

License GPL (>= 2)

URL https://kosukeimai.github.io/MatchIt/, https://github.com/kosukeimai/MatchIt

BugReports https://github.com/kosukeimai/MatchIt/issues

VignetteBuilder knitr

RoxygenNote 7.3.2

Config/testthat/edition 3

NeedsCompilation yes

Author Daniel Ho [aut] (ORCID: <https://orcid.org/0000-0002-2195-5469>),
Kosuke Imai [aut] (ORCID: <https://orcid.org/0000-0002-2748-1022>),
Gary King [aut] (ORCID: <https://orcid.org/0000-0002-5327-7631>),
Elizabeth Stuart [aut] (ORCID: <https://orcid.org/0000-0002-9042-8611>),
Alex Whitworth [ctb],
Noah Greifer [cre, aut] (ORCID:
<https://orcid.org/0000-0003-3067-7154>)

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

Repository CRAN

Date/Publication 2025-05-30 09:00:09 UTC

Contents

`add_s.weights` *Add sampling weights to a* `matchit` *object*

Description

Adds sampling weights to a `matchit` object so that they are incorporated into balance assessment and creation of the weights. This would typically only be used when an argument to `s.weights` was not supplied to `matchit()` (i.e., because they were not to be included in the estimation of the propensity score) but sampling weights are required for generalizing an effect to the correct population. Without adding sampling weights to the `matchit` object, balance assessment tools (i.e.,

summary.matchit() and plot.matchit()) will not calculate balance statistics correctly, and the weights produced by match_data() and get_matches() will not incorporate the sampling weights.

Usage

```
add_s.weights(m, s.weights = NULL, data = NULL)
```

Arguments

m	a matchit object; the output of a call to matchit(), typically with the s.weights argument unspecified.
s.weights	an numeric vector of sampling weights to be added to the matchit object. Can also be specified as a string containing the name of variable in data to be used or a one-sided formula with the variable on the right-hand side (e.g., ~ SW).
data	a data frame containing the sampling weights if given as a string or formula. If unspecified, add_s.weights() will attempt to find the dataset using the environment of the matchit object.

Value

a matchit object with an s.weights component containing the supplied sampling weights. If s.weights = NULL, the original matchit object is returned.

Author(s)

Noah Greifer

See Also

matchit(); match_data()

Examples

```
data("lalonde")

# Generate random sampling weights, just
# for this example
sw <- rchisq(nrow(lalonde), 2)

# NN PS match using logistic regression PS that doesn't
# include sampling weights
m.out <- matchit(treat ~ age + educ + race + nodegree +
                   married  + re74 + re75,
                 data = lalonde)

m.out

# Add s.weights to the matchit object
m.out <- add_s.weights(m.out, sw)

m.out #note additional output
```

```
# Check balance; note that sample sizes incorporate
# s.weights
summary(m.out, improvement = FALSE)
```

`distance` *Propensity scores and other distance measures*

Description

Several matching methods require or can involve the distance between treated and control units. Options include the Mahalanobis distance, propensity score distance, or distance between user-supplied values. Propensity scores are also used for common support via the `discard` options and for defining calipers. This page documents the options that can be supplied to the `distance` argument to `matchit()`.

Allowable options

There are four ways to specify the `distance` argument: 1) as a string containing the name of a method for estimating propensity scores, 2) as a string containing the name of a method for computing pairwise distances from the covariates, 3) as a vector of values whose pairwise differences define the distance between units, or 4) as a distance matrix containing all pairwise distances. The options are detailed below.

Propensity score estimation methods:

When `distance` is specified as the name of a method for estimating propensity scores (described below), a propensity score is estimated using the variables in `formula` and the method corresponding to the given argument. This propensity score can be used to compute the distance between units as the absolute difference between the propensity scores of pairs of units. Propensity scores can also be used to create calipers and common support restrictions, whether or not they are used in the actual distance measure used in the matching, if any.

In addition to the `distance` argument, two other arguments can be specified that relate to the estimation and manipulation of the propensity scores. The `link` argument allows for different links to be used in models that require them such as generalized linear models, for which the logit and probit links are allowed, among others. In addition to specifying the link, the `link` argument can be used to specify whether the propensity score or the linearized version of the propensity score should be used (i.e., the linear predictor of the propensity score model); by specifying `link = "linear.{link}"`, the linearized version will be used. When `link = "linear.logit"`, for example, this requests the logit of a propensity score estimated with a logistic link.

The `distance.options` argument can also be specified, which should be a list of values passed to the propensity score-estimating function, for example, to choose specific options or tuning parameters for the estimation method. If `formula`, `data`, or `verbose` are not supplied to `distance.options`, the corresponding arguments from `matchit()` will be automatically supplied. See the Examples for demonstrations of the uses of `link` and `distance.options`. When `s.weights` is supplied in the call to `matchit()`, it will automatically be passed to the propensity score-estimating function as the `weights` argument unless otherwise described below.

The following methods for estimating propensity scores are allowed:

`"glm"` The propensity scores are estimated using a generalized linear model (e.g., logistic regression). The `formula` supplied to `matchit()` is passed directly to `glm()`, and `predict.glm()` is used to compute the propensity scores. The `link` argument can be specified as a link function supplied to `binomial()`, e.g., `"logit"`, which is the default. When `link` is prepended by `"linear."`, the linear predictor is used instead of the predicted probabilities. `distance = "glm"` with `link = "logit"` (logistic regression) is the default in `matchit()`. (This used to be able to be requested as `distance = "ps"`, which still works.)

`"gam"` The propensity scores are estimated using a generalized additive model. The `formula` supplied to `matchit()` is passed directly to `mgcv::gam()`, and `mgcv::predict.gam()` is used to compute the propensity scores. The `link` argument can be specified as a link function supplied to `binomial()`, e.g., `"logit"`, which is the default. When `link` is prepended by `"linear."`, the linear predictor is used instead of the predicted probabilities. Note that unless the smoothing functions `mgcv::s()`, `mgcv::te()`, `mgcv::ti()`, or `mgcv::t2()` are used in `formula`, a generalized additive model is identical to a generalized linear model and will estimate the same propensity scores as `glm()`. See the documentation for `mgcv::gam()`, `mgcv::formula.gam()`, and `mgcv::gam.models()` for more information on how to specify these models. Also note that the formula returned in the `matchit()` output object will be a simplified version of the supplied formula with smoothing terms removed (but all named variables present).

`"gbm"` The propensity scores are estimated using a generalized boosted model. The `formula` supplied to `matchit()` is passed directly to `gbm::gbm()`, and `gbm::predict.gbm()` is used to compute the propensity scores. The optimal tree is chosen using 5-fold cross-validation by default, and this can be changed by supplying an argument to `method` to `distance.options`; see `gbm::gbm.perf()` for details. The `link` argument can be specified as `"linear"` to use the linear predictor instead of the predicted probabilities. No other links are allowed. The tuning parameter defaults differ from `gbm::gbm()`; they are as follows: `n.trees = 1e4`, `interaction.depth = 3`, `shrinkage = .01`, `bag.fraction = 1`, `cv.folds = 5`, `keep.data = FALSE`. These are the same defaults as used in *WeightIt* and *twang*, except for `cv.folds` and `keep.data`. Note this is not the same use of generalized boosted modeling as in *twang*; here, the number of trees is chosen based on cross-validation or out-of-bag error, rather than based on optimizing balance. **twang** should not be cited when using this method to estimate propensity scores. Note that because there is a random component to choosing the tuning parameter, results will vary across runs unless a seed is set.

`"lasso"`, `"ridge"`, `"elasticnet"` The propensity scores are estimated using a lasso, ridge, or elastic net model, respectively. The `formula` supplied to `matchit()` is processed with `model.matrix()` and passed to `glmnet::cv.glmnet()`, and `glmnet::predict.cv.glmnet()` is used to compute the propensity scores. The `link` argument can be specified as a link function supplied to `binomial()`, e.g., `"logit"`, which is the default. When `link` is prepended by `"linear."`, the linear predictor is used instead of the predicted probabilities. When `link = "log"`, a Poisson model is used. For `distance = "elasticnet"`, the `alpha` argument, which controls how to prioritize the lasso and ridge penalties in the elastic net, is set to .5 by default and can be changed by supplying an argument to `alpha` in `distance.options`. For `"lasso"` and `"ridge"`, `alpha` is set to 1 and 0, respectively, and cannot be changed. The `cv.glmnet()` defaults are used to select the tuning parameters and generate predictions and can be modified using `distance.options`. If the `s` argument is passed to `distance.options`, it will be passed to `predict.cv.glmnet()`. Note that because there is a random component to choosing the tuning parameter, results will vary across runs unless a seed is set.

`"rpart"` The propensity scores are estimated using a classification tree. The `formula` supplied

to matchit() is passed directly to rpart::rpart(), and rpart::predict.rpart() is used to compute the propensity scores. The link argument is ignored, and predicted probabilities are always returned as the distance measure.

"randomforest" The propensity scores are estimated using a random forest. The formula supplied to matchit() is passed directly to randomForest::randomForest(), and randomForest::predict.randomFo is used to compute the propensity scores. The link argument is ignored, and predicted probabilities are always returned as the distance measure. Note that because there is a random component, results will vary across runs unless a seed is set.

"nnet" The propensity scores are estimated using a single-hidden-layer neural network. The formula supplied to matchit() is passed directly to nnet::nnet(), and fitted() is used to compute the propensity scores. The link argument is ignored, and predicted probabilities are always returned as the distance measure. An argument to size must be supplied to distance.options when using method = "nnet".

"cbps" The propensity scores are estimated using the covariate balancing propensity score (CBPS) algorithm, which is a form of logistic regression where balance constraints are incorporated to a generalized method of moments estimation of of the model coefficients. The formula supplied to matchit() is passed directly to CBPS::CBPS(), and fitted() is used to compute the propensity scores. The link argument can be specified as "linear" to use the linear predictor instead of the predicted probabilities. No other links are allowed. The estimand argument supplied to matchit() will be used to select the appropriate estimand for use in defining the balance constraints, so no argument needs to be supplied to ATT in CBPS.

"bart" The propensity scores are estimated using Bayesian additive regression trees (BART). The formula supplied to matchit() is passed directly to dbarts::bart2(), and dbarts::fitted.bart() is used to compute the propensity scores. The link argument can be specified as "linear" to use the linear predictor instead of the predicted probabilities. When s.weights is supplied to matchit(), it will not be passed to bart2 because the weights argument in bart2 does not correspond to sampling weights. Note that because there is a random component to choosing the tuning parameter, results will vary across runs unless the seed argument is supplied to distance.options. Note that setting a seed using set.seed() is not sufficient to guarantee reproducibility unless single-threading is used. See dbarts::bart2() for details.

Methods for computing distances from covariates:

The following methods involve computing a distance matrix from the covariates themselves without estimating a propensity score. Calipers on the distance measure and common support restrictions cannot be used, and the distance component of the output object will be empty because no propensity scores are estimated. The link and distance.options arguments are ignored with these methods. See the individual matching methods pages for whether these distances are allowed and how they are used. Each of these distance measures can also be calculated outside matchit() using its corresponding function.

"euclidean" The Euclidean distance is the raw distance between units, computed as

$$d_{ij} = \sqrt{(x_i - x_j)(x_i - x_j)'}$$

It is sensitive to the scale of the covariates, so covariates with larger scales will take higher priority.

"scaled_euclidean" The scaled Euclidean distance is the Euclidean distance computed on the scaled (i.e., standardized) covariates. This ensures the covariates are on the same scale. The covariates are standardized using the pooled within-group standard deviations, computed by

treatment group-mean centering each covariate before computing the standard deviation in the full sample.

`"mahalanobis"` The Mahalanobis distance is computed as

$$d_{ij} = \sqrt{(x_i - x_j)\Sigma^{-1}(x_i - x_j)'}$$

where Σ is the pooled within-group covariance matrix of the covariates, computed by treatment group-mean centering each covariate before computing the covariance in the full sample. This ensures the variables are on the same scale and accounts for the correlation between covariates.

`"robust_mahalanobis"` The robust rank-based Mahalanobis distance is the Mahalanobis distance computed on the ranks of the covariates with an adjustment for ties. It is described in Rosenbaum (2010, ch. 8) as an alternative to the Mahalanobis distance that handles outliers and rare categories better than the standard Mahalanobis distance but is not affinely invariant.

To perform Mahalanobis distance matching *and* estimate propensity scores to be used for a purpose other than matching, the `mahvars` argument should be used along with a different specification to `distance`. See the individual matching method pages for details on how to use `mahvars`.

Distances supplied as a numeric vector or matrix:

`distance` can also be supplied as a numeric vector whose values will be taken to function like propensity scores; their pairwise difference will define the distance between units. This might be useful for supplying propensity scores computed outside `matchit()` or resupplying `matchit()` with propensity scores estimated previously without having to recompute them.

`distance` can also be supplied as a matrix whose values represent the pairwise distances between units. The matrix should either be a square, with a row and column for each unit (e.g., as the output of a call to `as.matrix([dist](.))`), or have as many rows as there are treated units and as many columns as there are control units (e.g., as the output of a call to `mahalanobis_dist()` or `optmatch::match_on()`). Distance values of `Inf` will disallow the corresponding units to be matched. When `distance` is a supplied as a numeric vector or matrix, `link` and `distance.options` are ignored.

Note

In versions of *MatchIt* prior to 4.0.0, `distance` was specified in a slightly different way. When specifying arguments using the old syntax, they will automatically be converted to the corresponding method in the new syntax but a warning will be thrown. `distance = "logit"`, the old default, will still work in the new syntax, though `distance = "glm", link = "logit"` is preferred (note that these are the default settings and don't need to be made explicit).

Examples

```
data("lalonde")

# Matching on logit of a PS estimated with logistic
# regression:
m.out1 <- matchit(treat ~ age + educ + race + married +
                    nodegree + re74 + re75,
                  data = lalonde,
                  distance = "glm",
```

```
                  link = "linear.logit")

# GAM logistic PS with smoothing splines (s()):
m.out2 <- matchit(treat ~ s(age) + s(educ) +
                    race + married +
                    nodegree + re74 + re75,
                  data = lalonde,
                  distance = "gam")
summary(m.out2$model)


# CBPS for ATC matching w/replacement, using the just-
# identified version of CBPS (setting method = "exact"):
m.out3 <- matchit(treat ~ age + educ + race + married +
                    nodegree + re74 + re75,
                  data = lalonde,
                  distance = "cbps",
                  estimand = "ATC",
                  distance.options = list(method = "exact"),
                  replace = TRUE)

# Mahalanobis distance matching - no PS estimated
m.out4 <- matchit(treat ~ age + educ + race + married +
                    nodegree + re74 + re75,
                  data = lalonde,
                  distance = "mahalanobis")

m.out4$distance #NULL

# Mahalanobis distance matching with PS estimated
# for use in a caliper; matching done on mahvars
m.out5 <- matchit(treat ~ age + educ + race + married +
                    nodegree + re74 + re75,
                  data = lalonde,
                  distance = "glm",
                  caliper = .1,
                  mahvars = ~ age + educ + race + married +
                                nodegree + re74 + re75)

summary(m.out5)

# User-supplied propensity scores
p.score <- fitted(glm(treat ~ age + educ + race + married +
                        nodegree + re74 + re75,
                      data = lalonde,
                      family = binomial))

m.out6 <- matchit(treat ~ age + educ + race + married +
                    nodegree + re74 + re75,
                  data = lalonde,
                  distance = p.score)

# User-supplied distance matrix using rank_mahalanobis()
```

```
dist_mat <- robust_mahalanobis_dist(
              treat ~ age + educ + race + nodegree +
                married + re74 + re75,
              data = lalonde)

m.out7 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  distance = dist_mat)
```

`lalonde` *Data from National Supported Work Demonstration and PSID, as analyzed by Dehejia and Wahba (1999).*

Description

This is a subsample of the data from the treated group in the National Supported Work Demonstration (NSW) and the comparison sample from the Population Survey of Income Dynamics (PSID). This data was previously analyzed extensively by Lalonde (1986) and Dehejia and Wahba (1999).

Format

A data frame with 614 observations (185 treated, 429 control). There are 9 variables measured for each individual.

- "treat" is the treatment assignment (1=treated, 0=control).
- "age" is age in years.
- "educ" is education in number of years of schooling.
- "race" is the individual's race/ethnicity, (Black, Hispanic, or White). Note previous versions of this dataset used indicator variables `black` and `hispan` instead of a single race variable.
- "married" is an indicator for married (1=married, 0=not married).
- "nodegree" is an indicator for whether the individual has a high school degree (1=no degree, 0=degree).
- "re74" is income in 1974, in U.S. dollars.
- "re75" is income in 1975, in U.S. dollars.
- "re78" is income in 1978, in U.S. dollars.

"treat" is the treatment variable, "re78" is the outcome, and the others are pre-treatment covariates.

References

Lalonde, R. (1986). Evaluating the econometric evaluations of training programs with experimental data. *American Economic Review* 76: 604-620.

Dehejia, R.H. and Wahba, S. (1999). Causal Effects in Nonexperimental Studies: Re-Evaluating the Evaluation of Training Programs. *Journal of the American Statistical Association* 94: 1053-1062.

mahalanobis_dist *Compute a Distance Matrix*

Description

The functions compute a distance matrix, either for a single dataset (i.e., the distances between all pairs of units) or for two groups defined by a splitting variable (i.e., the distances between all units in one group and all units in the other). These distance matrices include the Mahalanobis distance, Euclidean distance, scaled Euclidean distance, and robust (rank-based) Mahalanobis distance. These functions can be used as inputs to the `distance` argument to `matchit()` and are used to compute the corresponding distance matrices within `matchit()` when named.

Usage

```
mahalanobis_dist(
  formula = NULL,
  data = NULL,
  s.weights = NULL,
  var = NULL,
  discarded = NULL,
  ...
)

scaled_euclidean_dist(
  formula = NULL,
  data = NULL,
  s.weights = NULL,
  var = NULL,
  discarded = NULL,
  ...
)

robust_mahalanobis_dist(
  formula = NULL,
  data = NULL,
  s.weights = NULL,
  discarded = NULL,
  ...
)

euclidean_dist(formula = NULL, data = NULL, ...)
```

Arguments

`formula`	a formula with the treatment (i.e., splitting variable) on the left side and the covariates used to compute the distance matrix on the right side. If there is

	no left-hand-side variable, the distances will be computed between all pairs of units. If NULL, all the variables in data will be used as covariates.
data	a data frame containing the variables named in formula. If formula is NULL, all variables in data will be used as covariates.
s.weights	when var = NULL, an optional vector of sampling weights used to compute the variances used in the Mahalanobis, scaled Euclidean, and robust Mahalanobis distances.
var	for mahalanobis_dist(), a covariance matrix used to scale the covariates. For scaled_euclidean_dist(), either a covariance matrix (from which only the diagonal elements will be used) or a vector of variances used to scale the covariates. If NULL, these values will be calculated using formulas described in Details.
discarded	a logical vector denoting which units are to be discarded or not. This is used only when var = NULL. The scaling factors will be computed only using the non-discarded units, but the distance matrix will be computed for all units (discarded and non-discarded).
...	ignored. Included to make cycling through these functions easier without having to change the arguments supplied.

Details

The **Euclidean distance** (computed using euclidean_dist()) is the raw distance between units, computed as

$$d_{ij} = \sqrt{(x_i - x_j)(x_i - x_j)'}$$

where x_i and x_j are vectors of covariates for units i and j, respectively. The Euclidean distance is sensitive to the scales of the variables and their redundancy (i.e., correlation). It should probably not be used for matching unless all of the variables have been previously scaled appropriately or are already on the same scale. It forms the basis of the other distance measures.

The **scaled Euclidean distance** (computed using scaled_euclidean_dist()) is the Euclidean distance computed on the scaled covariates. Typically the covariates are scaled by dividing by their standard deviations, but any scaling factor can be supplied using the var argument. This leads to a distance measure computed as

$$d_{ij} = \sqrt{(x_i - x_j)S_d^{-1}(x_i - x_j)'}$$

where S_d is a diagonal matrix with the squared scaling factors on the diagonal. Although this measure is not sensitive to the scales of the variables (because they are all placed on the same scale), it is still sensitive to redundancy among the variables. For example, if 5 variables measure approximately the same construct (i.e., are highly correlated) and 1 variable measures another construct, the first construct will have 5 times as much influence on the distance between units as the second construct. The Mahalanobis distance attempts to address this issue.

The **Mahalanobis distance** (computed using mahalanobis_dist()) is computed as

$$d_{ij} = \sqrt{(x_i - x_j)S^{-1}(x_i - x_j)'}$$

where S is a scaling matrix, typically the covariance matrix of the covariates. It is essentially equivalent to the Euclidean distance computed on the scaled principal components of the covariates.

This is the most popular distance matrix for matching because it is not sensitive to the scale of the covariates and accounts for redundancy between them. The scaling matrix can also be supplied using the `var` argument.

The Mahalanobis distance can be sensitive to outliers and long-tailed or otherwise non-normally distributed covariates and may not perform well with categorical variables due to prioritizing rare categories over common ones. One solution is the rank-based **robust Mahalanobis distance** (computed using `robust_mahalanobis_dist()`), which is computed by first replacing the covariates with their ranks (using average ranks for ties) and rescaling each ranked covariate by a constant scaling factor before computing the usual Mahalanobis distance on the rescaled ranks.

The Mahalanobis distance and its robust variant are computed internally by transforming the covariates in such a way that the Euclidean distance computed on the scaled covariates is equal to the requested distance. For the Mahalanobis distance, this involves replacing the covariates vector x_i with $x_i S^{-.5}$, where $S^{-.5}$ is the Cholesky decomposition of the (generalized) inverse of the covariance matrix S.

When a left-hand-side splitting variable is present in `formula` and `var = NULL` (i.e., so that the scaling matrix is computed internally), the covariance matrix used is the "pooled" covariance matrix, which essentially is a weighted average of the covariance matrices computed separately within each level of the splitting variable to capture within-group variation and reduce sensitivity to covariate imbalance. This is also true of the scaling factors used in the scaled Euclidean distance.

Value

A numeric distance matrix. When `formula` has a left-hand-side (treatment) variable, the matrix will have one row for each treated unit and one column for each control unit. Otherwise, the matrix will have one row and one column for each unit.

Author(s)

Noah Greifer

References

Rosenbaum, P. R. (2010). *Design of observational studies*. Springer.

Rosenbaum, P. R., & Rubin, D. B. (1985). Constructing a Control Group Using Multivariate Matched Sampling Methods That Incorporate the Propensity Score. *The American Statistician*, 39(1), 33–38. doi:10.2307/2683903

Rubin, D. B. (1980). Bias Reduction Using Mahalanobis-Metric Matching. *Biometrics*, 36(2), 293–298. doi:10.2307/2529981

See Also

`distance`, `matchit()`, `dist()` (which is used internally to compute some Euclidean distances)

`optmatch::match_on()`, which provides similar functionality but with fewer options and a focus on efficient storage of the output.

Examples

```
data("lalonde")

# Computing the scaled Euclidean distance between all units:
d <- scaled_euclidean_dist(~ age + educ + race + married,
                           data = lalonde)

# Another interface using the data argument:
dat <- subset(lalonde, select = c(age, educ, race, married))
d <- scaled_euclidean_dist(data = dat)

# Computing the Mahalanobis distance between treated and
# control units:
d <- mahalanobis_dist(treat ~ age + educ + race + married,
                      data = lalonde)

# Supplying a covariance matrix or vector of variances (note:
# a bit more complicated with factor variables)
dat <- subset(lalonde, select = c(age, educ, married, re74))
vars <- sapply(dat, var)

d <- scaled_euclidean_dist(data = dat, var = vars)

# Same result:
d <- scaled_euclidean_dist(data = dat, var = diag(vars))

# Discard units:
discard <- sample(c(TRUE, FALSE), nrow(lalonde),
                  replace = TRUE, prob = c(.2, .8))

d <- mahalanobis_dist(treat ~ age + educ + race + married,
                      data = lalonde, discarded = discard)
dim(d) #all units present in distance matrix
table(lalonde$treat)
```

`matchit` *Matching for Causal Inference*

Description

`matchit()` is the main function of *MatchIt* and performs pairing, subset selection, and subclassification with the aim of creating treatment and control groups balanced on included covariates. *MatchIt* implements the suggestions of Ho, Imai, King, and Stuart (2007) for improving parametric statistical models by preprocessing data with nonparametric matching methods.

This page documents the overall use of `matchit()`, but for specifics of how `matchit()` works with individual matching methods, see the individual pages linked in the Details section below.

Usage

```
matchit(
  formula,
  data = NULL,
  method = "nearest",
  distance = "glm",
  link = "logit",
  distance.options = list(),
  estimand = "ATT",
  exact = NULL,
  mahvars = NULL,
  antiexact = NULL,
  discard = "none",
  reestimate = FALSE,
  s.weights = NULL,
  replace = FALSE,
  m.order = NULL,
  caliper = NULL,
  std.caliper = TRUE,
  ratio = 1,
  verbose = FALSE,
  include.obj = FALSE,
  normalize = TRUE,
  ...
)
```

Arguments

formula	a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the matching. This formula will be supplied to the functions that estimate the distance measure. The formula should be specified as A ~ X1 + X2 + ... where A represents the treatment variable and X1 and X2 are covariates.
data	a data frame containing the variables named in formula and possible other arguments. If not found in data, the variables will be sought in the environment.
method	the matching method to be used. The allowed methods are "nearest" for nearest neighbor matching (on the propensity score by default), "optimal" for optimal pair matching, "full" for optimal full matching, "quick" for generalized (quick) full matching, "genetic" for genetic matching, "cem" for coarsened exact matching, "exact" for exact matching, "cardinality" for cardinality and profile matching, and "subclass" for subclassification. When set to NULL, no matching will occur, but propensity score estimation and common support restrictions will still occur if requested. See the linked pages for each method for more details on what these methods do, how the arguments below are used by each on, and what additional arguments are allowed.
distance	the distance measure to be used. Can be either the name of a method of estimating propensity scores (e.g., "glm"), the name of a method of computing a

distance matrix from the covariates (e.g., "mahalanobis"), a vector of already-computed distance measures, or a matrix of pairwise distances. See distance for allowable options. The default is "glm" for propensity scores estimated with logistic regression using glm(). Ignored for some methods; see individual methods pages for information on whether and how the distance measure is used.

link — when distance is specified as a string, an additional argument controlling the link function used in estimating the distance measure. Allowable options depend on the specific distance value specified. See distance for allowable options with each option. The default is "logit", which, along with distance = "glm", identifies the default measure as logistic regression propensity scores.

distance.options — a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to distance. See distance for an example of its use.

estimand — a string containing the name of the target estimand desired. Can be one of "ATT", "ATC", or "ATE". Default is "ATT". See Details and the individual methods pages for information on how this argument is used.

exact — for methods that allow it, for which variables exact matching should take place. Can be specified as a string containing the names of variables in data to be used or a one-sided formula with the desired variables on the right-hand side (e.g., ~ X3 + X4). See the individual methods pages for information on whether and how this argument is used.

mahvars — for methods that allow it, on which variables Mahalanobis distance matching should take place when distance corresponds to propensity scores. Usually used to perform Mahalanobis distance matching within propensity score calipers, where the propensity scores are computed using formula and distance. Can be specified as a string containing the names of variables in data to be used or a one-sided formula with the desired variables on the right-hand side (e.g., ~ X3 + X4). See the individual methods pages for information on whether and how this argument is used.

antiexact — for methods that allow it, for which variables anti-exact matching should take place. Anti-exact matching ensures paired individuals do not have the same value of the anti-exact matching variable(s). Can be specified as a string containing the names of variables in data to be used or a one-sided formula with the desired variables on the right-hand side (e.g., ~ X3 + X4). See the individual methods pages for information on whether and how this argument is used.

discard — a string containing a method for discarding units outside a region of common support. When a propensity score is estimated or supplied to distance as a vector, the options are "none", "treated", "control", or "both". For "none", no units are discarded for common support. Otherwise, units whose propensity scores fall outside the corresponding region are discarded. Can also be a logical vector where TRUE indicates the unit is to be discarded. Default is "none" for no common support restriction. See Details.

reestimate — if discard is not "none" and propensity scores are estimated, whether to re-estimate the propensity scores in the remaining sample. Default is FALSE to use the propensity scores estimated in the original sample.

`s.weights` an optional numeric vector of sampling weights to be incorporated into propensity score models and balance statistics. Can also be specified as a string containing the name of variable in `data` to be used or a one-sided formula with the variable on the right-hand side (e.g., `~ SW`). Not all propensity score models accept sampling weights; see `distance` for information on which do and do not, and see `vignette("sampling-weights")` for details on how to use sampling weights in a matching analysis.

`replace` for methods that allow it, whether matching should be done with replacement (`TRUE`), where control units are allowed to be matched to several treated units, or without replacement (`FALSE`), where control units can only be matched to one treated unit each. See the individual methods pages for information on whether and how this argument is used. Default is `FALSE` for matching without replacement.

`m.order` for methods that allow it, the order that the matching takes place. Allowable options depend on the matching method. The default of `NULL` corresponds to `"largest"` when a propensity score is estimated or supplied as a vector and `"data"` otherwise.

`caliper` for methods that allow it, the width(s) of the caliper(s) to use in matching. Should be a numeric vector with each value named according to the variable to which the caliper applies. To apply to the distance measure, the value should be unnamed. See the individual methods pages for information on whether and how this argument is used. Positive values require the distance between paired units to be no larger than the supplied caliper; negative values require the distance between paired units to be larger than the absolute value value of the supplied caliper. The default is `NULL` for no caliper.

`std.caliper` `logical`; when a caliper is specified, whether the the caliper is in standard deviation units (`TRUE`) or raw units (`FALSE`). Can either be of length 1, applying to all calipers, or of length equal to the length of `caliper`. Default is `TRUE`.

`ratio` for methods that allow it, how many control units should be matched to each treated unit in k:1 matching. Should be a single integer value. See the individual methods pages for information on whether and how this argument is used. The default is 1 for 1:1 matching.

`verbose` `logical`; whether information about the matching process should be printed to the console. What is printed depends on the matching method. Default is `FALSE` for no printing other than warnings.

`include.obj` `logical`; whether to include any objects created in the matching process in the output, i.e., by the functions from other packages `matchit()` calls. What is included depends on the matching method. Default is `FALSE`.

`normalize` `logical`; whether to rescale the nonzero weights in each treatment group to have an average of 1. Default is `TRUE`. See "How Matching Weights Are Computed" below for more details.

`...` additional arguments passed to the functions used in the matching process. See the individual methods pages for information on what additional arguments are allowed for each method.

Details

Details for the various matching methods can be found at the following help pages:

- method_nearest for nearest neighbor matching
- method_optimal for optimal pair matching
- method_full for optimal full matching
- method_quick for generalized (quick) full matching
- method_genetic for genetic matching
- method_cem for coarsened exact matching
- method_exact for exact matching
- method_cardinality for cardinality and profile matching
- method_subclass for subclassification

The pages contain information on what the method does, which of the arguments above are allowed with them and how they are interpreted, and what additional arguments can be supplied to further tune the method. Note that the default method with no arguments supplied other than formula and data is 1:1 nearest neighbor matching without replacement on a propensity score estimated using a logistic regression of the treatment on the covariates. This is not the same default offered by other matching programs, such as those in *Matching*, teffects in Stata, or PROC PSMATCH in SAS, so care should be taken if trying to replicate the results of those programs.

When method = NULL, no matching will occur, but any propensity score estimation and common support restriction will. This can be a simple way to estimate the propensity score for use in future matching specifications without having to re-estimate it each time. The matchit() output with no matching can be supplied to summary() to examine balance prior to matching on any of the included covariates and on the propensity score if specified. All arguments other than distance, discard, and reestimate will be ignored.

See distance for details on the several ways to specify the distance, link, and distance.options arguments to estimate propensity scores and create distance measures.

When the treatment variable is not a 0/1 variable, it will be coerced to one and returned as such in the matchit() output (see section Value, below). The following rules are used: 1) if 0 is one of the values, it will be considered the control and the other value the treated; 2) otherwise, if the variable is a factor, levels(treat)[1] will be considered control and the other value the treated; 3) otherwise, sort(unique(treat))[1] will be considered control and the other value the treated. It is safest to ensure the treatment variable is a 0/1 variable.

The discard option implements a common support restriction. It can only be used when a distance measure is an estimated propensity score or supplied as a vector and is ignored for some matching methods. When specified as "treated", treated units whose distance measure is outside the range of distance measures of the control units will be discarded. When specified as "control", control units whose distance measure is outside the range of distance measures of the treated units will be discarded. When specified as "both", treated and control units whose distance measure is outside the intersection of the range of distance measures of the treated units and the range of distance measures of the control units will be discarded. When reestimate = TRUE and distance corresponds to a propensity score-estimating function, the propensity scores are re-estimated in the remaining units prior to being used for matching or calipers.

Caution should be used when interpreting effects estimated with various values of `estimand`. Setting `estimand = "ATT"` doesn't necessarily mean the average treatment effect in the treated is being estimated; it just means that for matching methods, treated units will be untouched and given weights of 1 and control units will be matched to them (and the opposite for `estimand = "ATC"`). If a caliper is supplied or treated units are removed for common support or some other reason (e.g., lacking matches when using exact matching), the actual estimand targeted is not the ATT but the treatment effect in the matched sample. The argument to `estimand` simply triggers which units are matched to which, and for stratification-based methods (exact matching, CEM, full matching, and subclassification), determines the formula used to compute the stratification weights.

How Matching Weights Are Computed:

Matching weights are computed in one of two ways depending on whether matching was done with replacement or not.

Matching without replacement and subclassification:

For matching *without* replacement (except for cardinality matching), including subclassification, each unit is assigned to a subclass, which represents the pair they are a part of (in the case of k:1 matching) or the stratum they belong to (in the case of exact matching, coarsened exact matching, full matching, or subclassification). The formula for computing the weights depends on the argument supplied to `estimand`. A new "stratum propensity score" (p_i^s) is computed for each unit i as $p_i^s = \frac{1}{n_s}\sum_{j:s_j=s_i} I(A_j = 1)$ where n_s is the size of subclass s and $I(A_j = 1)$ is 1 if unit j is treated and 0 otherwise. That is, the stratum propensity score for stratum s is the proportion of units in stratum s that are in the treated group, and all units in stratum s are assigned that stratum propensity score. This is distinct from the propensity score used for matching, if any. Weights are then computed using the standard formulas for inverse probability weights with the stratum propensity score inserted:

- for the ATT, weights are 1 for the treated units and $\frac{p^s}{1-p^s}$ for the control units
- for the ATC, weights are $\frac{1-p^s}{p^s}$ for the treated units and 1 for the control units
- for the ATE, weights are $\frac{1}{p^s}$ for the treated units and $\frac{1}{1-p^s}$ for the control units.

For cardinality matching, all matched units receive a weight of 1.

Matching with replacement:

For matching *with* replacement, units are not assigned to unique strata. For the ATT, each treated unit gets a weight of 1. Each control unit is weighted as the sum of the inverse of the number of control units matched to the same treated unit across its matches. For example, if a control unit was matched to a treated unit that had two other control units matched to it, and that same control was matched to a treated unit that had one other control unit matched to it, the control unit in question would get a weight of $1/3 + 1/2 = 5/6$. For the ATC, the same is true with the treated and control labels switched. The weights are computed using the `match.matrix` component of the `matchit()` output object.

Normalized weights:

When `normalize = TRUE` (the default), in each treatment group, weights are divided by the mean of the nonzero weights in that treatment group to make the weights sum to the number of units in that treatment group (i.e., to have an average of 1).

Sampling weights:

If sampling weights are included through the `s.weights` argument, they will be included in the `matchit()` output object but not incorporated into the matching weights. `match_data()`, which extracts the matched set from a `matchit` object, combines the matching weights and sampling weights.

Value

When method is something other than "subclass", a matchit object with the following components:

match.matrix	a matrix containing the matches. The row names correspond to the treated units and the values in each row are the names (or indices) of the control units matched to each treated unit. When treated units are matched to different numbers of control units (e.g., with variable ratio matching or matching with a caliper), empty spaces will be filled with NA. Not included when method is "full", "cem" (unless k2k = TRUE), "exact", "quick", or "cardinality" (unless mahvars is supplied and ratio is an integer).
subclass	a factor containing matching pair/stratum membership for each unit. Unmatched units will have a value of NA. Not included when replace = TRUE or when method = "cardinality" unless mahvars is supplied and ratio is an integer.
weights	a numeric vector of estimated matching weights. Unmatched and discarded units will have a weight of zero.
model	the fit object of the model used to estimate propensity scores when distance is specified as a method of estimating propensity scores. When reestimate = TRUE, this is the model estimated after discarding units.
X	a data frame of covariates mentioned in formula, exact, mahvars, caliper, and antiexact.
call	the matchit() call.
info	information on the matching method and distance measures used.
estimand	the argument supplied to estimand.
formula	the formula supplied.
treat	a vector of treatment status converted to zeros (0) and ones (1) if not already in that format.
distance	a vector of distance values (i.e., propensity scores) when distance is supplied as a method of estimating propensity scores or a numeric vector.
discarded	a logical vector denoting whether each observation was discarded (TRUE) or not (FALSE) by the argument to discard.
s.weights	the vector of sampling weights supplied to the s.weights argument, if any.
exact	a one-sided formula containing the variables, if any, supplied to exact.
mahvars	a one-sided formula containing the variables, if any, supplied to mahvars.
obj	when include.obj = TRUE, an object containing the intermediate results of the matching procedure. See the individual methods pages for what this component will contain.

When method = "subclass", a matchit.subclass object with the same components as above except that match.matrix is excluded and one additional component, q.cut, is included, containing a vector of the distance measure cutpoints used to define the subclasses. See method_subclass for details.

Author(s)

Daniel Ho, Kosuke Imai, Gary King, and Elizabeth Stuart wrote the original package. Starting with version 4.0.0, Noah Greifer is the primary maintainer and developer.

References

Ho, D. E., Imai, K., King, G., & Stuart, E. A. (2007). Matching as Nonparametric Preprocessing for Reducing Model Dependence in Parametric Causal Inference. *Political Analysis*, 15(3), 199–236. doi:10.1093/pan/mpl013

Ho, D. E., Imai, K., King, G., & Stuart, E. A. (2011). MatchIt: Nonparametric Preprocessing for Parametric Causal Inference. *Journal of Statistical Software*, 42(8). doi:10.18637/jss.v042.i08

See Also

`summary.matchit()` for balance assessment after matching, `plot.matchit()` for plots of covariate balance and propensity score overlap after matching.

- `vignette("MatchIt")` for an introduction to matching with *MatchIt*
- `vignette("matching-methods")` for descriptions of the variety of matching methods and options available
- `vignette("assessing-balance")` for information on assessing the quality of a matching specification
- `vignette("estimating-effects")` for instructions on how to estimate treatment effects after matching
- `vignette("sampling-weights")` for a guide to using *MatchIt* with sampling weights.

Examples

```
data("lalonde")

# Default: 1:1 NN PS matching w/o replacement

m.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde)
m.out1
summary(m.out1)

# 1:1 NN Mahalanobis distance matching w/ replacement and
# exact matching on married and race

m.out2 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  distance = "mahalanobis",
                  replace = TRUE,
                  exact = ~ married + race)
m.out2
summary(m.out2, un = TRUE)
```

```
# 2:1 NN Mahalanobis distance matching within caliper defined
# by a probit pregression PS

m.out3 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  distance = "glm",
                  link = "probit",
                  mahvars = ~ age + educ + re74 + re75,
                  caliper = .1,
                  ratio = 2)
m.out3
summary(m.out3, un = TRUE)

# Optimal full PS matching for the ATE within calipers on
# PS, age, and educ

m.out4 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "full",
                  estimand = "ATE",
                  caliper = c(.1, age = 2, educ = 1),
                  std.caliper = c(TRUE, FALSE, FALSE))
m.out4
summary(m.out4, un = TRUE)

# Subclassification on a logistic PS with 10 subclasses after
# discarding controls outside common support of PS

s.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "subclass",
                  distance = "glm",
                  discard = "control",
                  subclass = 10)
s.out1
summary(s.out1, un = TRUE)
```

`match_data` *Construct a matched dataset from a* `matchit` *object*

Description

`match_data()` and `get_matches()` create a data frame with additional variables for the distance measure, matching weights, and subclasses after matching. This dataset can be used to estimate treatment effects after matching or subclassification. `get_matches()` is most useful after matching with replacement; otherwise, `match_data()` is more flexible. See Details below for the difference between them.

Usage

```
match_data(
  object,
  group = "all",
  distance = "distance",
  weights = "weights",
  subclass = "subclass",
  data = NULL,
  include.s.weights = TRUE,
  drop.unmatched = TRUE
)

match.data(...)

get_matches(
  object,
  distance = "distance",
  weights = "weights",
  subclass = "subclass",
  id = "id",
  data = NULL,
  include.s.weights = TRUE
)
```

Arguments

object	a matchit object; the output of a call to matchit().
group	which group should comprise the matched dataset: "all" for all units, "treated" for just treated units, or "control" for just control units. Default is "all".
distance	a string containing the name that should be given to the variable containing the distance measure in the data frame output. Default is "distance", but "prop.score" or similar might be a good alternative if propensity scores were used in matching. Ignored if a distance measure was not supplied or estimated in the call to matchit().
weights	a string containing the name that should be given to the variable containing the matching weights in the data frame output. Default is "weights".
subclass	a string containing the name that should be given to the variable containing the subclasses or matched pair membership in the data frame output. Default is "subclass".
data	a data frame containing the original dataset to which the computed output variables (distance, weights, and/or subclass) should be appended. If empty, match_data() and get_matches() will attempt to find the dataset using the environment of the matchit object, which can be unreliable; see Notes.
include.s.weights	logical; whether to multiply the estimated weights by the sampling weights supplied to matchit(), if any. Default is TRUE. If FALSE, the weights in the

match_data() or get_matches() output should be multiplied by the sampling weights before being supplied to the function estimating the treatment effect in the matched data.

drop.unmatched logical; whether the returned data frame should contain all units (FALSE) or only units that were matched (i.e., have a matching weight greater than zero) (TRUE). Default is TRUE to drop unmatched units.

... arguments passed to match_data().

id a string containing the name that should be given to the variable containing the unit IDs in the data frame output. Default is "id". Only used with get_matches(); for match_data(), the units IDs are stored in the row names of the returned data frame.

Details

match_data() creates a dataset with one row per unit. It will be identical to the dataset supplied except that several new columns will be added containing information related to the matching. When drop.unmatched = TRUE, the default, units with weights of zero, which are those units that were discarded by common support or the caliper or were simply not matched, will be dropped from the dataset, leaving only the subset of matched units. The idea is for the output of match_data() to be used as the dataset input in calls to glm() or similar to estimate treatment effects in the matched sample. It is important to include the weights in the estimation of the effect and its standard error. The subclass column, when created, contains pair or subclass membership and should be used to estimate the effect and its standard error. Subclasses will only be included if there is a subclass component in the matchit object, which does not occur with matching with replacement, in which case get_matches() should be used. See vignette("estimating-effects") for information on how to use match_data() output to estimate effects. match.data() is an alias for match_data().

get_matches() is similar to match_data(); the primary difference occurs when matching is performed with replacement, i.e., when units do not belong to a single matched pair. In this case, the output of get_matches() will be a dataset that contains one row per unit for each pair they are a part of. For example, if matching was performed with replacement and a control unit was matched to two treated units, that control unit will have two rows in the output dataset, one for each pair it is a part of. Weights are computed for each row, and, for control units, are equal to the inverse of the number of control units in each control unit's subclass; treated units get a weight of 1. Unmatched units are dropped. An additional column with unit IDs will be created (named using the id argument) to identify when the same unit is present in multiple rows. This dataset structure allows for the inclusion of both subclass membership and repeated use of units, unlike the output of match_data(), which lacks subclass membership when matching is done with replacement. A match.matrix component of the matchit object must be present to use get_matches(); in some forms of matching, it is absent, in which case match_data() should be used instead. See vignette("estimating-effects") for information on how to use get_matches() output to estimate effects after matching with replacement.

Value

A data frame containing the data supplied in the data argument or in the original call to matchit() with the computed output variables appended as additional columns, named according the arguments above. For match_data(), the group and drop.unmatched arguments control whether only

subsets of the data are returned. See Details above for how match_data() and get_matches() differ. Note that get_matches sorts the data by subclass and treatment status, unlike match_data(), which uses the order of the data.

The returned data frame will contain the variables in the original data set or dataset supplied to data and the following columns:

distance	The propensity score, if estimated or supplied to the distance argument in matchit() as a vector.
weights	The computed matching weights. These must be used in effect estimation to correctly incorporate the matching.
subclass	Matching strata membership. Units with the same value are in the same stratum.
id	The ID of each unit, corresponding to the row names in the original data or dataset supplied to data. Only included in get_matches output. This column can be used to identify which rows belong to the same unit since the same unit may appear multiple times if reused in matching with replacement.

These columns will take on the name supplied to the corresponding arguments in the call to match_data() or get_matches(). See Examples for an example of rename the distance column to "prop.score".

If data or the original dataset supplied to matchit() was a data.table or tbl, the match_data() output will have the same class, but the get_matches() output will always be a base R data.frame.

In addition to their base class (e.g., data.frame or tbl), returned objects have the class matchdata or getmatches. This class is important when using rbind() to append matched datasets.

Note

The most common way to use match_data() and get_matches() is by supplying just the matchit object, e.g., as match_data(m.out). A data set will first be searched in the environment of the matchit formula, then in the calling environment of match_data() or get_matches(), and finally in the model component of the matchit object if a propensity score was estimated.

When called from an environment different from the one in which matchit() was originally called and a propensity score was not estimated (or was but with discard not "none" and reestimate = TRUE), this syntax may not work because the original dataset used to construct the matched dataset will not be found. This can occur when matchit() was run within an lapply() or purrr::map() call. The solution, which is recommended in all cases, is simply to supply the original dataset to the data argument of match_data(), e.g., as match_data(m.out, data = original_data), as demonstrated in the Examples.

See Also

matchit(); rbind.matchdata()

vignette("estimating-effects") for uses of match_data() and get_matches() in estimating treatment effects.

Examples

```
data("lalonde")
```

```
# 4:1 matching w/replacement
m.out1 <- matchit(treat ~ age + educ + married +
                    race + nodegree + re74 + re75,
                  data = lalonde,
                  replace = TRUE,
                  caliper = .05,
                  ratio = 4)

m.data1 <- match_data(m.out1,
                      data = lalonde,
                      distance = "prop.score")
dim(m.data1) #one row per matched unit
head(m.data1, 10)

g.matches1 <- get_matches(m.out1,
                          data = lalonde,
                          distance = "prop.score")
dim(g.matches1) #multiple rows per matched unit
head(g.matches1, 10)
```

`method_cardinality` *Cardinality Matching*

Description

In `matchit()`, setting `method = "cardinality"` performs cardinality matching and other forms of matching that use mixed integer programming. Rather than forming pairs, cardinality matching selects the largest subset of units that satisfies user-supplied balance constraints on mean differences. One of several available optimization programs can be used to solve the mixed integer program. The default is the HiGHS library as implemented in the *highs* package, both of which are free, but performance can be improved using Gurobi and the *gurobi* package, for which there is a free academic license.

This page details the allowable arguments with `method = "cardinality"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for cardinality matching:

```
matchit(formula,
        data = NULL,
        method = "cardinality",
        estimand = "ATT",
        exact = NULL,
        mahvars = NULL,
        s.weights = NULL,
        ratio = 1,
        verbose = FALSE,
        tols = .05,
        std.tols = TRUE,
```

```
          solver = "highs",
          ...)
```

Arguments

formula	a two-sided formula object containing the treatment and covariates to be balanced.
data	a data frame containing the variables named in formula. If not found in data, the variables will be sought in the environment.
method	set here to "cardinality".
estimand	a string containing the desired estimand. Allowable options include "ATT", "ATC", and "ATE". See Details.
exact	for which variables exact matching should take place. Separate optimization will occur within each subgroup of the exact matching variables.
mahvars	which variables should be used for pairing after subset selection. Can only be set when ratio is a whole number. See Details.
s.weights	the variable containing sampling weights to be incorporated into the optimization. The balance constraints refer to the product of the sampling weights and the matching weights, and the sum of the product of the sampling and matching weights will be maximized.
ratio	the desired ratio of control to treated units. Can be set to NA to maximize sample size without concern for this ratio. See Details.
verbose	logical; whether information about the matching process should be printed to the console.
...	additional arguments that control the matching specification:

tols numeric; a vector of imbalance tolerances for mean differences, one for each covariate in formula. If only one value is supplied, it is applied to all. See std.tols below. Default is .05 for standardized mean differences of at most .05 for all covariates between the treatment groups in the matched sample.

std.tols logical; whether each entry in tols corresponds to a raw or standardized mean difference. If only one value is supplied, it is applied to all. Default is TRUE for standardized mean differences. The standardization factor is the pooled standard deviation when estimand = "ATE", the standard deviation of the treated group when estimand = "ATT", and the standard deviation of the control group when estimand = "ATC" (the same as used in summary.matchit()).

solver the name of solver to use to solve the optimization problem. Available options include "highs", "glpk", "symphony", and "gurobi" for HiGHS (implemented in the *highs* package), GLPK (implemented in the *Rglpk* package), SYMPHONY (implemented in the *Rsymphony* package), and Gurobi (implemented in the *gurobi* package), respectively. The differences between them are in speed and solving ability. HiGHS (the default) and GLPK are the easiest to install, but Gurobi is recommended as it consistently outperforms other solvers and can find solutions even when others

can't, and in less time. Gurobi is proprietary but can be used with a free trial or academic license. SYMPHONY may not produce reproducible results, even with a seed set.

`time` the maximum amount of time before the optimization routine aborts, in seconds. Default is 120 (2 minutes). For large problems, this should be set much higher.

The arguments `distance` (and related arguments), `replace`, `m.order`, and `caliper` (and related arguments) are ignored with a warning.

Details

Cardinality and Profile Matching:

Two types of matching are available with `method = "cardinality"`: cardinality matching and profile matching.

Cardinality matching finds the largest matched set that satisfies the balance constraints between treatment groups, with the additional constraint that the ratio of the number of matched control to matched treated units is equal to `ratio` (1 by default), mimicking k:1 matching. When not all treated units are included in the matched set, the estimand no longer corresponds to the ATT, so cardinality matching should be avoided if retaining the ATT is desired. To request cardinality matching, `estimand` should be set to `"ATT"` or `"ATC"` and `ratio` should be set to a positive integer. 1:1 cardinality matching is the default method when no arguments are specified.

Profile matching finds the largest matched set that satisfies balance constraints between each treatment group and a specified target sample. When `estimand = "ATT"`, it will find the largest subset of the control units that satisfies the balance constraints with respect to the treated group, which is left intact. When `estimand = "ATE"`, it will find the largest subsets of the treated group and of the control group that are balanced to the overall sample. To request profile matching for the ATT, `estimand` should be set to `"ATT"` and `ratio` to `NA`. To request profile matching for the ATE, `estimand` should be set to `"ATE"` and `ratio` can be set either to `NA` to maximize the size of each sample independently or to a positive integer to ensure that the ratio of matched control units to matched treated treats is fixed, mimicking k:1 matching. Unlike cardinality matching, profile matching retains the requested estimand if a solution is found.

Neither method involves creating pairs in the matched set, but it is possible to perform an additional round of pairing within the matched sample after cardinality matching or profile matching for the ATE with a fixed whole number sample size ratio by supplying the desired pairing variables to `mahvars`. Doing so will trigger optimal matching using `optmatch::pairmatch()` on the Mahalanobis distance computed using the variables supplied to `mahvars`. The balance or composition of the matched sample will not change, but additional precision and robustness can be gained by forming the pairs.

The weights are scaled so that the sum of the weights in each group is equal to the number of matched units in the smaller group when cardinality matching or profile matching for the ATE, and scaled so that the sum of the weights in the control group is equal to the number of treated units when profile matching for the ATT. When the sample sizes of the matched groups is the same (i.e., when `ratio = 1`), no scaling is done. Robust standard errors should be used in effect estimation after cardinality or profile matching (and cluster-robust standard errors if additional pairing is done in the matched sample). See `vignette("estimating-effects")` for more information.

Specifying Balance Constraints:

The balance constraints are on the (standardized) mean differences between the matched treatment groups for each covariate. Balance constraints should be set by supplying arguments to `tols` and `std.tols`. For example, setting `tols = .1` and `std.tols = TRUE` requests that all the mean differences in the matched sample should be within .1 standard deviations for each covariate. Different tolerances can be set for different variables; it might be beneficial to constrain the mean differences for highly prognostic covariates more tightly than for other variables. For example, one could specify `tols = c(.001, .05), std.tols = c(TRUE, FALSE)` to request that the standardized mean difference for the first covariate is less than .001 and the raw mean difference for the second covariate is less than .05. The values should be specified in the order they appear in `formula`, except when interactions are present. One can run the following code:

```
MatchIt:::get_assign(model.matrix(~X1*X2 + X3, data = data))[-1]
```

which will output a vector of numbers and the variable to which each number corresponds; the first entry in `tols` corresponds to the variable labeled 1, the second to the variable labeled 2, etc.

Dealing with Errors and Warnings:

When the optimization cannot be solved at all, or at least within the time frame specified in the argument to `time`, an error or warning will appear. Unfortunately, it is hard to know exactly the cause of the failure and what measures should be taken to rectify it.

A warning that says `"The optimizer failed to find an optimal solution in the time alotted. The returned solution may not be optimal."` usually means that an optimal solution may be possible to find with more time, in which case `time` should be increased or a faster solver should be used. Even with this warning, a potentially usable solution will be returned, so don't automatically take it to mean the optimization failed. Sometimes, when there are multiple solutions with the same resulting sample size, the optimizers will stall at one of them, not thinking it has found the optimum. The result should be checked to see if it can be used as the solution.

An error that says `"The optimization problem may be infeasible."` usually means that there is a issue with the optimization problem, i.e., that there is no possible way to satisfy the constraints. To rectify this, one can try relaxing the constraints by increasing the value of `tols` or use another solver. Sometimes Gurobi can solve problems that the other solvers cannot.

Outputs

Most outputs described in `matchit()` are returned with `method = "cardinality"`. Unless `mahvars` is specified, the `match.matrix` and `subclass` components are omitted because no pairing or subclassification is done. When `include.obj = TRUE` in the call to `matchit()`, the output of the optimization function will be included in the output. When `exact` is specified, this will be a list of such objects, one for each stratum of the exact variables.

References

In a manuscript, you should reference the solver used in the optimization. For example, a sentence might read:

Cardinality matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R with the optimization performed by HiGHS (Huangfu & Hall, 2018).

See `vignette("matching-methods")` for more literature on cardinality matching.

See Also

matchit() for a detailed explanation of the inputs and outputs of a call to matchit().

designmatch, which performs cardinality and profile matching with many more options and more flexibility. The implementations of cardinality matching differ between *MatchIt* and *designmatch*, so their results might differ.

optweight, which offers similar functionality but in the context of weighting rather than matching.

Examples

```
data("lalonde")

#Choose your solver; "gurobi" is best, "highs" is free and
#easy to install
solver <- "highs"

m.out1 <- matchit(treat ~ age + educ + re74,
                  data = lalonde,
                  method = "cardinality",
                  estimand = "ATT",
                  ratio = 1,
                  tols = .2,
                  solver = solver)
m.out1
summary(m.out1)

# Profile matching for the ATT
m.out2 <- matchit(treat ~ age + educ + re74,
                  data = lalonde,
                  method = "cardinality",
                  estimand = "ATT",
                  ratio = NA,
                  tols = .2,
                  solver = solver)
m.out2
summary(m.out2, un = FALSE)

# Profile matching for the ATE
m.out3 <- matchit(treat ~ age + educ + re74,
                  data = lalonde,
                  method = "cardinality",
                  estimand = "ATE",
                  ratio = NA,
                  tols = .2,
                  solver = solver)
m.out3
summary(m.out3, un = FALSE)


# Pairing after 1:1 cardinality matching:
m.out1b <- matchit(treat ~ age + educ + re74,
                   data = lalonde,
```

```
                    method = "cardinality",
                    estimand = "ATT",
                    ratio = 1,
                    tols = .15,
                    solver = solver,
                    mahvars = ~ age + educ + re74)

# Note that balance doesn't change but pair distances
# are lower for the paired-upon variables
summary(m.out1b, un = FALSE)
summary(m.out1, un = FALSE)

# In these examples, a high tol was used and
# few covariate matched on in order to not take too long;
# with real data, tols should be much lower and more
# covariates included if possible.
```

`method_cem` *Coarsened Exact Matching*

Description

In `matchit()`, setting `method = "cem"` performs coarsened exact matching. With coarsened exact matching, covariates are coarsened into bins, and a complete cross of the coarsened covariates is used to form subclasses defined by each combination of the coarsened covariate levels. Any subclass that doesn't contain both treated and control units is discarded, leaving only subclasses containing treatment and control units that are exactly equal on the coarsened covariates. The coarsening process can be controlled by an algorithm or by manually specifying cutpoints and groupings. The benefits of coarsened exact matching are that the tradeoff between exact matching and approximate balancing can be managed to prevent discarding too many units, which can otherwise occur with exact matching.

This page details the allowable arguments with `method = "cem"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for coarsened exact matching:

```
matchit(formula,
        data = NULL,
        method = "cem",
        estimand = "ATT",
        s.weights = NULL,
        verbose = FALSE,
        ...)
```

Arguments

`formula`	a two-sided formula object containing the treatment and covariates to be used in creating the subclasses defined by a full cross of the coarsened covariate levels.

`data` a data frame containing the variables named in `formula`. If not found in `data`, the variables will be sought in the environment.

`method` set here to `"cem"`.

`estimand` a string containing the desired estimand. Allowable options include `"ATT"`, `"ATC"`, and `"ATE"`. The estimand controls how the weights are computed; see the Computing Weights section at `matchit()` for details. When `k2k = TRUE` (see below), `estimand` also controls how the matching is done.

`s.weights` the variable containing sampling weights to be incorporated into balance statistics or the scaling factors when `k2k = TRUE` and certain methods are used.

`verbose` `logical`; whether information about the matching process should be printed to the console.

`...` additional arguments to control the matching process.

`grouping` a named list with an (optional) entry for each categorical variable to be matched on. Each element should itself be a list, and each entry of the sublist should be a vector containing levels of the variable that should be combined to form a single level. Any categorical variables not included in `grouping` will remain as they are in the data, which means exact matching, with no coarsening, will take place on these variables. See Details.

`cutpoints` a named list with an (optional) entry for each numeric variable to be matched on. Each element describes a way of coarsening the corresponding variable. They can be a vector of cutpoints that demarcate bins, a single number giving the number of bins, or a string corresponding to a method of computing the number of bins. Allowable strings include `"sturges"`, `"scott"`, and `"fd"`, which use the functions `grDevices::nclass.Sturges()`, `grDevices::nclass.scott()`, and `grDevices::nclass.FD()`, respectively. The default is `"sturges"` for variables that are not listed or if no argument is supplied. Can also be a single value to be applied to all numeric variables. See Details.

`k2k` `logical`; whether 1:1 matching should occur within the matched strata. If `TRUE` nearest neighbor matching without replacement will take place within each stratum, and any unmatched units will be dropped (e.g., if there are more treated than control units in the stratum, the treated units without a match will be dropped). The `k2k.method` argument controls how the distance between units is calculated.

`k2k.method` `character`; how the distance between units should be calculated if `k2k = TRUE`. Allowable arguments include `NULL` (for random matching), any argument to `distance()` for computing a distance matrix from covariates (e.g., `"mahalanobis"`), or any allowable argument to `method` in `dist()`. Matching will take place on the original (non-coarsened) variables. The default is `"mahalanobis"`.

`mpower` if `k2k.method = "minkowski"`, the power used in creating the distance. This is passed to the `p` argument of `dist()`.

`m.order` `character`; the order that the matching takes place when `k2k = TRUE`. Allowable options include `"closest"`, where matching takes place in ascending order of the smallest distance between units; `"farthest"`, where matching takes place in descending order of the smallest distance between

units; "random", where matching takes place in a random order; and "data" where matching takes place based on the order of units in the data. When m.order = "random", results may differ across different runs of the same code unless a seed is set and specified with set.seed(). The default of NULL corresponds to "data". See method_nearest for more information.

The arguments distance (and related arguments), exact, mahvars, discard (and related arguments), replace, caliper (and related arguments), and ratio are ignored with a warning.

Details

If the coarsening is such that there are no exact matches with the coarsened variables, the grouping and cutpoints arguments can be used to modify the matching specification. Reducing the number of cutpoints or grouping some variable values together can make it easier to find matches. See Examples below. Removing variables can also help (but they will likely not be balanced unless highly correlated with the included variables). To take advantage of coarsened exact matching without failing to find any matches, the covariates can be manually coarsened outside of matchit() and then supplied to the exact argument in a call to matchit() with another matching method.

Setting k2k = TRUE is equivalent to first doing coarsened exact matching with k2k = FALSE and then supplying stratum membership as an exact matching variable (i.e., in exact) to another call to matchit() with method = "nearest". It is also equivalent to performing nearest neighbor matching supplying coarsened versions of the variables to exact, except that method = "cem" automatically coarsens the continuous variables. The estimand argument supplied with method = "cem" functions the same way it would in these alternate matching calls, i.e., by determining the "focal" group that controls the order of the matching.

Grouping and Cutpoints:

The grouping and cutpoints arguments allow one to fine-tune the coarsening of the covariates. grouping is used for combining categories of categorical covariates and cutpoints is used for binning numeric covariates. The values supplied to these arguments should be iteratively changed until a matching solution that balances covariate balance and remaining sample size is obtained. The arguments are described below.

grouping*:*
The argument to grouping must be a list, where each component has the name of a categorical variable, the levels of which are to be combined. Each component must itself be a list; this list contains one or more vectors of levels, where each vector corresponds to the levels that should be combined into a single category. For example, if a variable amount had levels "none", "some", and "a lot", one could enter grouping = list(amount = list(c("none"), c("some", "a lot"))), which would group "some" and "a lot" into a single category and leave "none" in its own category. Any levels left out of the list for each variable will be left alone (so c("none") could have been omitted from the previous code). Note that if a categorical variable does not appear in grouping, it will not be coarsened, so exact matching will take place on it. grouping should not be used for numeric variables with more than a few values; use cutpoints, described below, instead.

cutpoints*:*
The argument to cutpoints must also be a list, where each component has the name of a numeric variables that is to be binned. (As a shortcut, it can also be a single value that will be applied to all numeric variables). Each component can take one of three forms: a vector of

cutpoints that separate the bins, a single number giving the number of bins, or a string corresponding to an algorithm used to compute the number of bins. Any values at a boundary will be placed into the higher bin; e.g., if the cutpoints were `c(0, 5, 10)`, values of 5 would be placed into the same bin as values of 6, 7, 8, or 9, and values of 10 would be placed into a different bin. Internally, values of `-Inf` and `Inf` are appended to the beginning and end of the range. When given as a single number defining the number of bins, the bin boundaries are the maximum and minimum values of the variable with bin boundaries evenly spaced between them, i.e., not quantiles. A value of 0 will not perform any binning (equivalent to exact matching on the variable), and a value of 1 will remove the variable from the exact matching variables but it will be still used for pair matching when `k2k = TRUE`. The allowable strings include `"sturges"`, `"scott"`, and `"fd"`, which use the corresponding binning method, and `"q#"` where # is a number, which splits the variable into # equally-sized bins (i.e., quantiles).
An example of a way to supply an argument to `cutpoints` would be the following:

```
cutpoints = list(X1 = 4,
                 X2 = c(1.7, 5.5, 10.2),
                 X3 = "scott",
                 X4 = "q5")
```

This would split `X1` into 4 bins, `X2` into bins based on the provided boundaries, `X3` into a number of bins determined by `grDevices::nclass.scott()`, and `X4` into quintiles. All other numeric variables would be split into a number of bins determined by `grDevices::nclass.Sturges()`, the default.

Outputs

All outputs described in `matchit()` are returned with `method = "cem"` except for `match.matrix`. When `k2k = TRUE`, a `match.matrix` component with the matched pairs is also included. `include.obj` is ignored.

Note

This method does not rely on the *cem* package, instead using code written for *MatchIt*, but its design is based on the original *cem* functions. Versions of *MatchIt* prior to 4.1.0 did rely on *cem*, so results may differ between versions. There are a few differences between the ways *MatchIt* and *cem* (and older versions of *MatchIt*) differ in executing coarsened exact matching, described below.

- In *MatchIt*, when a single number is supplied to `cutpoints`, it describes the number of bins; in *cem*, it describes the number of cutpoints separating bins. The *MatchIt* method is closer to how `hist()` processes breaks points to create bins.
- In *MatchIt*, values on the cutpoint boundaries will be placed into the higher bin; in *cem*, they are placed into the lower bin. To avoid consequences of this choice, ensure the bin boundaries do not coincide with observed values of the variables.
- When `cutpoints` are used, `"ss"` (for Shimazaki-Shinomoto's rule) can be used in *cem* but not in *MatchIt*.
- When `k2k = TRUE`, *MatchIt* matches on the original variables (scaled), whereas *cem* matches on the coarsened variables. Because the variables are already exactly matched on the coarsened variables, matching in *cem* is equivalent to random matching within strata.
- When `k2k = TRUE`, in *MatchIt* matched units are identified by pair membership, and the original stratum membership prior to 1:1 matching is discarded. In *cem*, pairs are not identified beyond the stratum the members are part of.

- When `k2k = TRUE`, `k2k.method = "mahalanobis"` can be requested in *MatchIt* but not in *cem*.

References

In a manuscript, you don't need to cite another package when using `method = "cem"` because the matching is performed completely within *MatchIt*. For example, a sentence might read:

Coarsened exact matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R.

It would be a good idea to cite the following article, which develops the theory behind coarsened exact matching:

Iacus, S. M., King, G., & Porro, G. (2012). Causal Inference without Balance Checking: Coarsened Exact Matching. *Political Analysis*, 20(1), 1–24. doi:10.1093/pan/mpr013

See Also

`matchit()` for a detailed explanation of the inputs and outputs of a call to `matchit()`.

The *cem* package, upon which this method is based and which provided the workhorse in previous versions of *MatchIt*.

`method_exact` for exact matching, which performs exact matching on the covariates without coarsening.

Examples

```
data("lalonde")

# Coarsened exact matching on age, race, married, and educ with educ
# coarsened into 5 bins and race coarsened into 2 categories,
# grouping "white" and "hispan" together
cutpoints <- list(educ = 5)
grouping <- list(race = list(c("white", "hispan"),
                             c("black")))

m.out1 <- matchit(treat ~ age + race + married + educ,
                  data = lalonde,
                  method = "cem",
                  cutpoints = cutpoints,
                  grouping = grouping)
m.out1
summary(m.out1)

# The same but requesting 1:1 Mahalanobis distance matching with
# the k2k and k2k.method argument. Note the remaining number of units
# is smaller than when retaining the full matched sample.
m.out2 <- matchit(treat ~ age + race + married + educ,
                  data = lalonde,
                  method = "cem",
                  cutpoints = cutpoints,
                  grouping = grouping,
                  k2k = TRUE,
                  k2k.method = "mahalanobis")
```

```
m.out2
summary(m.out2, un = FALSE)
```

`method_exact` *Exact Matching*

Description

In `matchit()`, setting `method = "exact"` performs exact matching. With exact matching, a complete cross of the covariates is used to form subclasses defined by each combination of the covariate levels. Any subclass that doesn't contain both treated and control units is discarded, leaving only subclasses containing treatment and control units that are exactly equal on the included covariates. The benefits of exact matching are that confounding due to the covariates included is completely eliminated, regardless of the functional form of the treatment or outcome models. The problem is that typically many units will be discarded, sometimes dramatically reducing precision and changing the target population of inference. To use exact matching in combination with another matching method (i.e., to exact match on some covariates and some other form of matching on others), use the `exact` argument with that method.

This page details the allowable arguments with `method = "exact"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for exact matching:

```
matchit(formula,
        data = NULL,
        method = "exact",
        estimand = "ATT",
        s.weights = NULL,
        verbose = FALSE,
        ...)
```

Arguments

`formula`	a two-sided formula object containing the treatment and covariates to be used in creating the subclasses defined by a full cross of the covariate levels.
`data`	a data frame containing the variables named in `formula`. If not found in `data`, the variables will be sought in the environment.
`method`	set here to `"exact"`.
`estimand`	a string containing the desired estimand. Allowable options include `"ATT"`, `"ATC"`, and `"ATE"`. The estimand controls how the weights are computed; see the Computing Weights section at `matchit()` for details.
`s.weights`	the variable containing sampling weights to be incorporated into balance statistics. These weights do not affect the matching process.
`verbose`	`logical`; whether information about the matching process should be printed to the console.

`...` ignored.

The arguments `distance` (and related arguments), `exact`, `mahvars`, `discard` (and related arguments), `replace`, `m.order`, `caliper` (and related arguments), and `ratio` are ignored with a warning.

Outputs

All outputs described in `matchit()` are returned with `method = "exact"` except for `match.matrix`. This is because matching strata are not indexed by treated units as they are in some other forms of matching. `include.obj` is ignored.

References

In a manuscript, you don't need to cite another package when using `method = "exact"` because the matching is performed completely within *MatchIt*. For example, a sentence might read:

Exact matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R.

See Also

`matchit()` for a detailed explanation of the inputs and outputs of a call to `matchit()`. The `exact` argument can be used with other methods to perform exact matching in combination with other matching methods.

method_cem for coarsened exact matching, which performs exact matching on coarsened versions of the covariates.

Examples

```
data("lalonde")

# Exact matching on age, race, married, and educ
m.out1 <- matchit(treat ~ age + race +
                    married + educ,
                  data = lalonde,
                  method = "exact")
m.out1
summary(m.out1)
```

---

method_full *Optimal Full Matching*

Description

In `matchit()`, setting `method = "full"` performs optimal full matching, which is a form of subclassification wherein all units, both treatment and control (i.e., the "full" sample), are assigned to a subclass and receive at least one match. The matching is optimal in the sense that that sum of the absolute distances between the treated and control units in each subclass is as small as possible. The method relies on and is a wrapper for `optmatch::fullmatch()`.

Advantages of optimal full matching include that the matching order is not required to be specified, units do not need to be discarded, and it is less likely that extreme within-subclass distances will be large, unlike with standard subclassification. The primary output of full matching is a set of matching weights that can be applied to the matched sample; in this way, full matching can be seen as a robust alternative to propensity score weighting, robust in the sense that the propensity score model does not need to be correct to estimate the treatment effect without bias. Note: with large samples, the optimization may fail or run very slowly; one can try using `method = "quick"` instead, which also performs full matching but can be much faster.

This page details the allowable arguments with `method = "full"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for optimal full matching:

```
matchit(formula,
        data = NULL,
        method = "full",
        distance = "glm",
        link = "logit",
        distance.options = list(),
        estimand = "ATT",
        exact = NULL,
        mahvars = NULL,
        anitexact = NULL,
        discard = "none",
        reestimate = FALSE,
        s.weights = NULL,
        caliper = NULL,
        std.caliper = TRUE,
        verbose = FALSE,
        ...)
```

Arguments

`formula` a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the matching. This formula will be supplied to the functions that estimate the distance measure.

`data` a data frame containing the variables named in `formula`. If not found in `data`, the variables will be sought in the environment.

`method` set here to `"full"`.

`distance` the distance measure to be used. See `distance` for allowable options. Can be supplied as a distance matrix.

`link` when `distance` is specified as a method of estimating propensity scores, an additional argument controlling the link function used in estimating the distance measure. See `distance` for allowable options with each option.

`distance.options` a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to `distance`.

`estimand` a string containing the desired estimand. Allowable options include `"ATT"`, `"ATC"`, and `"ATE"`. The estimand controls how the weights are computed; see the Computing Weights section at `matchit()` for details.

`exact` for which variables exact matching should take place.

`mahvars` for which variables Mahalanobis distance matching should take place when `distance` corresponds to a propensity score (e.g., for caliper matching or to discard units for common support). If specified, the distance measure will not be used in matching.

`antiexact` for which variables anti-exact matching should take place. Anti-exact matching is processed using `optmatch::antiExactMatch()`.

`discard` a string containing a method for discarding units outside a region of common support. Only allowed when `distance` corresponds to a propensity score.

`reestimate` if `discard` is not `"none"`, whether to re-estimate the propensity score in the remaining sample prior to matching.

`s.weights` the variable containing sampling weights to be incorporated into propensity score models and balance statistics.

`caliper` the width(s) of the caliper(s) used for caliper matching. Calipers are processed by `optmatch::caliper()`. Positive and negative calipers are allowed. See Notes and Examples.

`std.caliper` `logical`; when calipers are specified, whether they are in standard deviation units (`TRUE`) or raw units (`FALSE`).

`verbose` `logical`; whether information about the matching process should be printed to the console.

`...` additional arguments passed to `optmatch::fullmatch()`. Allowed arguments include `min.controls`, `max.controls`, `omit.fraction`, `mean.controls`, `tol`, and `solver`. See the `optmatch::fullmatch()` documentation for details. In general, `tol` should be set to a low number (e.g., `1e-7`) to get a more precise solution.

The arguments `replace`, `m.order`, and `ratio` are ignored with a warning.

Details

Mahalanobis Distance Matching:

Mahalanobis distance matching can be done one of two ways:

1. If no propensity score needs to be estimated, `distance` should be set to `"mahalanobis"`, and Mahalanobis distance matching will occur using all the variables in `formula`. Arguments to `discard` and `mahvars` will be ignored, and a caliper can only be placed on named variables. For example, to perform simple Mahalanobis distance matching, the following could be run:

   ```
   matchit(treat ~ X1 + X2, method = "nearest",
           distance = "mahalanobis")
   ```

   With this code, the Mahalanobis distance is computed using `X1` and `X2`, and matching occurs on this distance. The `distance` component of the `matchit()` output will be empty.

2. If a propensity score needs to be estimated for any reason, e.g., for common support with `discard` or for creating a caliper, `distance` should be whatever method is used to estimate

the propensity score or a vector of distance measures, i.e., it should not be "mahalanobis". Use mahvars to specify the variables used to create the Mahalanobis distance. For example, to perform Mahalanobis within a propensity score caliper, the following could be run:

```
matchit(treat ~ X1 + X2 + X3, method = "nearest",
        distance =  "glm", caliper = .25,
        mahvars = ~ X1 + X2)
```

With this code, X1, X2, and X3 are used to estimate the propensity score (using the "glm" method, which by default is logistic regression), which is used to create a matching caliper. The actual matching occurs on the Mahalanobis distance computed only using X1 and X2, which are supplied to mahvars. Units whose propensity score difference is larger than the caliper will not be paired, and some treated units may therefore not receive a match. The estimated propensity scores will be included in the distance component of the matchit() output. See Examples.

Outputs

All outputs described in matchit() are returned with method = "full" except for match.matrix. This is because matching strata are not indexed by treated units as they are in some other forms of matching. When include.obj = TRUE in the call to matchit(), the output of the call to optmatch::fullmatch() will be included in the output. When exact is specified, this will be a list of such objects, one for each stratum of the exact variables.

Note

Calipers can only be used when min.controls is left at its default.

The option "optmatch_max_problem_size" is automatically set to Inf during the matching process, different from its default in *optmatch*. This enables matching problems of any size to be run, but may also let huge, infeasible problems get through and potentially take a long time or crash R. See optmatch::setMaxProblemSize() for more details.

References

In a manuscript, be sure to cite the following paper if using matchit() with method = "full":

Hansen, B. B., & Klopfer, S. O. (2006). Optimal Full Matching and Related Designs via Network Flows. *Journal of Computational and Graphical Statistics*, 15(3), 609–627. doi:10.1198/106186006X137047

For example, a sentence might read:

Optimal full matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R, which calls functions from the optmatch package (Hansen & Klopfer, 2006).

Theory is also developed in the following article:

Hansen, B. B. (2004). Full Matching in an Observational Study of Coaching for the SAT. Journal of the American Statistical Association, 99(467), 609–618. doi:10.1198/016214504000000647

See Also

matchit() for a detailed explanation of the inputs and outputs of a call to matchit().

optmatch::fullmatch(), which is the workhorse.

method_optimal for optimal pair matching, which is a special case of optimal full matching, and which relies on similar machinery. Results from method = "optimal" can be replicated with method = "full" by setting min.controls, max.controls, and mean.controls to the desired ratio.

method_quick for fast generalized quick matching, which is very similar to optimal full matching but can be dramatically faster at the expense of optimality and is less customizable.

Examples

```
data("lalonde")

# Optimal full PS matching
m.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "full")
m.out1
summary(m.out1)

# Optimal full Mahalanobis distance matching within a PS caliper
m.out2 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "full",
                  caliper = .01,
                  mahvars = ~ age + educ + re74 + re75)
m.out2
summary(m.out2, un = FALSE)

# Optimal full Mahalanobis distance matching within calipers
# of 500 on re74 and re75
m.out3 <- matchit(treat ~ age + educ + re74 + re75,
                  data = lalonde,
                  distance = "mahalanobis",
                  method = "full",
                  caliper = c(re74 = 500,
                              re75 = 500),
                  std.caliper = FALSE)
m.out3
summary(m.out3,
        addlvariables = ~race + nodegree + married,
        data = lalonde,
        un = FALSE)
```

---

method_genetic *Genetic Matching*

Description

In matchit(), setting method = "genetic" performs genetic matching. Genetic matching is a form of nearest neighbor matching where distances are computed as the generalized Mahalanobis distance, which is a generalization of the Mahalanobis distance with a scaling factor for each covariate that represents the importance of that covariate to the distance. A genetic algorithm is used to select the scaling factors. The scaling factors are chosen as those which maximize a criterion related to covariate balance, which can be chosen, but which by default is the smallest p-value in covariate balance tests among the covariates. This method relies on and is a wrapper for Matching::GenMatch() and Matching::Match(), which use rgenoud::genoud() to perform the optimization using the genetic algorithm.

This page details the allowable arguments with method = "genetic". See matchit() for an explanation of what each argument means in a general context and how it can be specified.

Below is how matchit() is used for genetic matching:

```
matchit(formula,
        data = NULL,
        method = "genetic",
        distance = "glm",
        link = "logit",
        distance.options = list(),
        estimand = "ATT",
        exact = NULL,
        mahvars = NULL,
        antiexact = NULL,
        discard = "none",
        reestimate = FALSE,
        s.weights = NULL,
        replace = FALSE,
        m.order = NULL,
        caliper = NULL,
        ratio = 1,
        verbose = FALSE,
        ...)
```

Arguments

formula	a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the matching. This formula will be supplied to the functions that estimate the distance measure and is used to determine the covariates whose balance is to be optimized.
data	a data frame containing the variables named in formula. If not found in data, the variables will be sought in the environment.
method	set here to "genetic".
distance	the distance measure to be used. See distance for allowable options. When set to a method of estimating propensity scores or a numeric vector of distance values, the distance measure is included with the covariates in formula to be

supplied to the generalized Mahalanobis distance matrix unless `mahvars` is specified. Otherwise, only the covariates in `formula` are supplied to the generalized Mahalanobis distance matrix to have their scaling factors chosen. `distance` *cannot* be supplied as a distance matrix. Supplying any method of computing a distance matrix (e.g., `"mahalanobis"`) has the same effect of omitting propensity score but does not affect how the distance between units is computed otherwise.

`link` when `distance` is specified as a method of estimating propensity scores, an additional argument controlling the link function used in estimating the distance measure. See `distance` for allowable options with each option.

`distance.options`
a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to `distance`.

`estimand` a string containing the desired estimand. Allowable options include `"ATT"` and `"ATC"`. See Details.

`exact` for which variables exact matching should take place.

`mahvars` when a distance corresponds to a propensity score (e.g., for caliper matching or to discard units for common support), which covariates should be supplied to the generalized Mahalanobis distance matrix for matching. If unspecified, all variables in `formula` will be supplied to the distance matrix. Use `mahvars` to only supply a subset. Even if `mahvars` is specified, balance will be optimized on all covariates in `formula`. See Details.

`antiexact` for which variables anti-exact matching should take place. Anti-exact matching is processed using the `restrict` argument to `Matching::GenMatch()` and `Matching::Match()`.

`discard` a string containing a method for discarding units outside a region of common support. Only allowed when `distance` corresponds to a propensity score.

`reestimate` if `discard` is not `"none"`, whether to re-estimate the propensity score in the remaining sample prior to matching.

`s.weights` the variable containing sampling weights to be incorporated into propensity score models and balance statistics. These are also supplied to `GenMatch()` for use in computing the balance t-test p-values in the process of matching.

`replace` whether matching should be done with replacement.

`m.order` the order that the matching takes place. Allowable options include `"largest"`, where matching takes place in descending order of distance measures; `"smallest"`, where matching takes place in ascending order of distance measures; `"random"`, where matching takes place in a random order; and `"data"` where matching takes place based on the order of units in the data. When `m.order = "random"`, results may differ across different runs of the same code unless a seed is set and specified with `set.seed()`. The default of `NULL` corresponds to `"largest"` when a propensity score is estimated or supplied as a vector and `"data"` otherwise.

`caliper` the width(s) of the caliper(s) used for caliper matching. See Details and Examples.

`std.caliper` `logical`; when calipers are specified, whether they are in standard deviation units (`TRUE`) or raw units (`FALSE`).

`ratio` how many control units should be matched to each treated unit for k:1 matching. Should be a single integer value.

`verbose` `logical`; whether information about the matching process should be printed to the console. When `TRUE`, output from `GenMatch()` with `print.level = 2` will be displayed. Default is `FALSE` for no printing other than warnings.

`...` additional arguments passed to `Matching::GenMatch()`. Potentially useful options include `pop.size`, `max.generations`, and `fit.func`. If `pop.size` is not specified, a warning from *Matching* will be thrown reminding you to change it. Note that the `ties` and `CommonSupport` arguments are set to `FALSE` and cannot be changed. If `distance.tolerance` is not specified, it is set to 0, whereas the default in *Matching* is 1e-5.

Details

In genetic matching, covariates play three roles: 1) as the variables on which balance is optimized, 2) as the variables in the generalized Mahalanobis distance between units, and 3) in estimating the propensity score. Variables supplied to `formula` are always used for role (1), as the variables on which balance is optimized. When `distance` corresponds to a propensity score, the covariates are also used to estimate the propensity score (unless it is supplied). When `mahvars` is specified, the named variables will form the covariates that go into the distance matrix. Otherwise, the variables in `formula` along with the propensity score will go into the distance matrix. This leads to three ways to use `distance` and `mahvars` to perform the matching:

1. When `distance` corresponds to a propensity score and `mahvars` *is not* specified, the covariates in `formula` along with the propensity score are used to form the generalized Mahalanobis distance matrix. This is the default and most typical use of `method = "genetic"` in `matchit()`.
2. When `distance` corresponds to a propensity score and `mahvars` *is* specified, the covariates in `mahvars` are used to form the generalized Mahalanobis distance matrix. The covariates in `formula` are used to estimate the propensity score and have their balance optimized by the genetic algorithm. The propensity score is not included in the generalized Mahalanobis distance matrix.
3. When `distance` is a method of computing a distance matrix (e.g.,`"mahalanobis"`), no propensity score is estimated, and the covariates in `formula` are used to form the generalized Mahalanobis distance matrix. Which specific method is supplied has no bearing on how the distance matrix is computed; it simply serves as a signal to omit estimation of a propensity score.

When a caliper is specified, any variables mentioned in `caliper`, possibly including the propensity score, will be added to the matching variables used to form the generalized Mahalanobis distance matrix. This is because *Matching* doesn't allow for the separation of caliper variables and matching variables in genetic matching.

Estimand:

The `estimand` argument controls whether control units are selected to be matched with treated units (`estimand = "ATT"`) or treated units are selected to be matched with control units (`estimand = "ATC"`). The "focal" group (e.g., the treated units for the ATT) is typically made to be the smaller treatment group, and a warning will be thrown if it is not set that way unless `replace = TRUE`.

Setting `estimand = "ATC"` is equivalent to swapping all treated and control labels for the treatment variable. When `estimand = "ATC"`, the default `m.order` is `"smallest"`, and the `match.matrix` component of the output will have the names of the control units as the rownames and be filled with the names of the matched treated units (opposite to when `estimand = "ATT"`). Note that the argument supplied to `estimand` doesn't necessarily correspond to the estimand actually targeted; it is merely a switch to trigger which treatment group is considered "focal". Note that while `GenMatch()` and `Match()` support the ATE as an estimand, `matchit()` only supports the ATT and ATC for genetic matching.

Reproducibility:

Genetic matching involves a random component, so a seed must be set using `set.seed()` to ensure reproducibility. When `cluster` is used for parallel processing, the seed must be compatible with parallel processing (e.g., by setting `kind = "L'Ecuyer-CMRG"`).

Outputs

All outputs described in `matchit()` are returned with `method = "genetic"`. When `replace = TRUE`, the `subclass` component is omitted. When `include.obj = TRUE` in the call to `matchit()`, the output of the call to `Matching::GenMatch()` will be included in the output.

References

In a manuscript, be sure to cite the following papers if using `matchit()` with `method = "genetic"`:

Diamond, A., & Sekhon, J. S. (2013). Genetic matching for estimating causal effects: A general multivariate matching method for achieving balance in observational studies. Review of Economics and Statistics, 95(3), 932–945. doi:10.1162/REST_a_00318

Sekhon, J. S. (2011). Multivariate and Propensity Score Matching Software with Automated Balance Optimization: The Matching package for R. Journal of Statistical Software, 42(1), 1–52. doi:10.18637/jss.v042.i07

For example, a sentence might read:

Genetic matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R, which calls functions from the Matching package (Diamond & Sekhon, 2013; Sekhon, 2011).

See Also

`matchit()` for a detailed explanation of the inputs and outputs of a call to `matchit()`.

`Matching::GenMatch()` and `Matching::Match()`, which do the work.

Examples

```
data("lalonde")

# 1:1 genetic matching with PS as a covariate
m.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "genetic",
                  pop.size = 10) #use much larger pop.size
m.out1
```

```
summary(m.out1)

# 2:1 genetic matching with replacement without PS
m.out2 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "genetic",
                  replace = TRUE,
                  ratio = 2,
                  distance = "mahalanobis",
                  pop.size = 10) #use much larger pop.size
m.out2
summary(m.out2, un = FALSE)

# 1:1 genetic matching on just age, educ, re74, and re75
# within calipers on PS and educ; other variables are
# used to estimate PS
m.out3 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "genetic",
                  mahvars = ~ age + educ + re74 + re75,
                  caliper = c(.05, educ = 2),
                  std.caliper = c(TRUE, FALSE),
                  pop.size = 10) #use much larger pop.size
m.out3
summary(m.out3, un = FALSE)
```

`method_nearest` *Nearest Neighbor Matching*

Description

In `matchit()`, setting `method = "nearest"` performs greedy nearest neighbor matching. A distance is computed between each treated unit and each control unit, and, one by one, each treated unit is assigned a control unit as a match. The matching is "greedy" in the sense that there is no action taken to optimize an overall criterion; each match is selected without considering the other matches that may occur subsequently.

This page details the allowable arguments with `method = "nearest"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for nearest neighbor matching:

```
matchit(formula,
        data = NULL,
        method = "nearest",
        distance = "glm",
        link = "logit",
```

```
          distance.options = list(),
          estimand = "ATT",
          exact = NULL,
          mahvars = NULL,
          antiexact = NULL,
          discard = "none",
          reestimate = FALSE,
          s.weights = NULL,
          replace = TRUE,
          m.order = NULL,
          caliper = NULL,
          ratio = 1,
          min.controls = NULL,
          max.controls = NULL,
          verbose = FALSE,
          ...)
```

Arguments

formula	a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the matching.
data	a data frame containing the variables named in formula. If not found in data, the variables will be sought in the environment.
method	set here to "nearest".
distance	the distance measure to be used. See distance for allowable options. Can be supplied as a distance matrix.
link	when distance is specified as a method of estimating propensity scores, an additional argument controlling the link function used in estimating the distance measure. See distance for allowable options with each option.
distance.options	a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to distance.
estimand	a string containing the desired estimand. Allowable options include "ATT" and "ATC". See Details.
exact	for which variables exact matching should take place; two units with different values of an exact matching variable will not be paired.
mahvars	for which variables Mahalanobis distance matching should take place when distance corresponds to a propensity score (e.g., for caliper matching or to discard units for common support). If specified, the distance measure will not be used in matching.
antiexact	for which variables anti-exact matching should take place; two units with the same value of an anti-exact matching variable will not be paired.
discard	a string containing a method for discarding units outside a region of common support. Only allowed when distance corresponds to a propensity score.
reestimate	if discard is not "none", whether to re-estimate the propensity score in the remaining sample prior to matching.

`s.weights` the variable containing sampling weights to be incorporated into propensity score models and balance statistics.

`replace` whether matching should be done with replacement (i.e., whether control units can be used as matches multiple times). See also the `reuse.max` argument below. Default is `FALSE` for matching without replacement.

`m.order` the order that the matching takes place. Allowable options include `"largest"`, where matching takes place in descending order of distance measures; `"smallest"`, where matching takes place in ascending order of distance measures; `"closest"`, where matching takes place in ascending order of the smallest distance between units; `"farthest"`, where matching takes place in descending order of the smallest distance between units; `"random"`, where matching takes place in a random order; and `"data"` where matching takes place based on the order of units in the data. When `m.order = "random"`, results may differ across different runs of the same code unless a seed is set and specified with `set.seed()`. The default of `NULL` corresponds to `"largest"` when a propensity score is estimated or supplied as a vector and `"data"` otherwise. See Details for more information.

`caliper` the width(s) of the caliper(s) used for caliper matching. Two units with a difference on a caliper variable larger than the caliper will not be paired. See Details and Examples.

`std.caliper` `logical`; when calipers are specified, whether they are in standard deviation units (`TRUE`) or raw units (`FALSE`).

`ratio` how many control units should be matched to each treated unit for k:1 matching. For variable ratio matching, see section "Variable Ratio Matching" in Details below. When `ratio` is greater than 1, all treated units will be attempted to be matched with a control unit before any treated unit is matched with a second control unit, etc. This reduces the possibility that control units will be used up before some treated units receive any matches.

`min.controls`, `max.controls`
for variable ratio matching, the minimum and maximum number of controls units to be matched to each treated unit. See section "Variable Ratio Matching" in Details below.

`verbose` `logical`; whether information about the matching process should be printed to the console. When `TRUE`, a progress bar implemented using *RcppProgress* will be displayed along with an estimate of the time remaining.

`...` additional arguments that control the matching specification:

`reuse.max` `numeric`; the maximum number of times each control can be used as a match. Setting `reuse.max = 1` corresponds to matching without replacement (i.e., `replace = FALSE`), and setting `reuse.max = Inf` corresponds to traditional matching with replacement (i.e., `replace = TRUE`) with no limit on the number of times each control unit can be matched. Other values restrict the number of times each control can be matched when matching with replacement. `replace` is ignored when `reuse.max` is specified.

`unit.id` one or more variables containing a unit ID for each observation, i.e., in case multiple observations correspond to the same unit. Once a control observation has been matched, no other observation with the same unit ID can be used as matches. This ensures each control unit is used only once even

if it has multiple observations associated with it. Omitting this argument is the same as giving each observation a unique ID.

Details

Mahalanobis Distance Matching:

Mahalanobis distance matching can be done one of two ways:

1. If no propensity score needs to be estimated, `distance` should be set to `"mahalanobis"`, and Mahalanobis distance matching will occur using all the variables in `formula`. Arguments to `discard` and `mahvars` will be ignored, and a caliper can only be placed on named variables. For example, to perform simple Mahalanobis distance matching, the following could be run:

   ```
   matchit(treat ~ X1 + X2, method = "nearest",
           distance = "mahalanobis")
   ```

   With this code, the Mahalanobis distance is computed using `X1` and `X2`, and matching occurs on this distance. The `distance` component of the `matchit()` output will be empty.

2. If a propensity score needs to be estimated for any reason, e.g., for common support with `discard` or for creating a caliper, `distance` should be whatever method is used to estimate the propensity score or a vector of distance measures. Use `mahvars` to specify the variables used to create the Mahalanobis distance. For example, to perform Mahalanobis within a propensity score caliper, the following could be run:

   ```
   matchit(treat ~ X1 + X2 + X3, method = "nearest",
           distance = "glm", caliper = .25,
           mahvars = ~ X1 + X2)
   ```

   With this code, `X1`, `X2`, and `X3` are used to estimate the propensity score (using the `"glm"` method, which by default is logistic regression), which is used to create a matching caliper. The actual matching occurs on the Mahalanobis distance computed only using `X1` and `X2`, which are supplied to `mahvars`. Units whose propensity score difference is larger than the caliper will not be paired, and some treated units may therefore not receive a match. The estimated propensity scores will be included in the `distance` component of the `matchit()` output. See Examples.

Estimand:

The `estimand` argument controls whether control units are selected to be matched with treated units (`estimand = "ATT"`) or treated units are selected to be matched with control units (`estimand = "ATC"`). The "focal" group (e.g., the treated units for the ATT) is typically made to be the smaller treatment group, and a warning will be thrown if it is not set that way unless `replace = TRUE`. Setting `estimand = "ATC"` is equivalent to swapping all treated and control labels for the treatment variable. When `estimand = "ATC"`, the default `m.order` is `"smallest"`, and the `match.matrix` component of the output will have the names of the control units as the rownames and be filled with the names of the matched treated units (opposite to when `estimand = "ATT"`). Note that the argument supplied to `estimand` doesn't necessarily correspond to the estimand actually targeted; it is merely a switch to trigger which treatment group is considered "focal".

Variable Ratio Matching:

`matchit()` can perform variable ratio "extremal" matching as described by Ming and Rosenbaum (2000; doi:10.1111/j.0006341X.2000.00118.x). This method tends to result in better balance than fixed ratio matching at the expense of some precision. When `ratio` > 1, rather than requiring all

treated units to receive `ratio` matches, each treated unit is assigned a value that corresponds to the number of control units they will be matched to. These values are controlled by the arguments `min.controls` and `max.controls`, which correspond to α and β, respectively, in Ming and Rosenbaum (2000), and trigger variable ratio matching to occur. Some treated units will receive `min.controls` matches and others will receive `max.controls` matches (and one unit may have an intermediate number of matches); how many units are assigned each number of matches is determined by the algorithm described in Ming and Rosenbaum (2000, p119). `ratio` controls how many total control units will be matched: `n1 * ratio` control units will be matched, where `n1` is the number of treated units, yielding the same total number of matched controls as fixed ratio matching does.

Variable ratio matching cannot be used with Mahalanobis distance matching or when `distance` is supplied as a matrix. The calculations of the numbers of control units each treated unit will be matched to occurs without consideration of `caliper` or `discard`. `ratio` does not have to be an integer but must be greater than 1 and less than `n0/n1`, where `n0` and `n1` are the number of control and treated units, respectively. Setting `ratio = n0/n1` performs a crude form of full matching where all control units are matched. If `min.controls` is not specified, it is set to 1 by default. `min.controls` must be less than `ratio`, and `max.controls` must be greater than `ratio`. See Examples below for an example of their use.

Using `m.order = "closest"` or `"farthest"`:

`m.order` can be set to `"closest"` or `"farthest"`, which work regardless of how the distance measure is specified. This matches in order of the distance between units. First, all the closest match is found for all treated units and the pairwise distances computed; when `m.order = "closest"` the pair with the smallest of the distances is matched first, and when `m.order = "farthest"`, the pair with the largest of the distances is matched first. Then, the pair with the second smallest (or largest) is matched second. If the matched control is ineligible (i.e., because it has already been used in a prior match), a new match is found for the treated unit, the new pair's distance is re-computed, and the pairs are re-ordered by distance.

Using `m.order = "closest"` ensures that the best possible matches are given priority, and in that sense should perform similarly to `m.order = "smallest"`. It can be used to ensure the best matches, especially when matching with a caliper. Using `m.order = "farthest"` ensures that the hardest units to match are given their best chance to find a close match, and in that sense should perform similarly to `m.order = "largest"`. It can be used to reduce the possibility of extreme imbalance when there are hard-to-match units competing for controls. Note that `m.order = "farthest"` **does not** implement "far matching" (i.e., finding the farthest control unit from each treated unit); it defines the order in which the closest matches are selected.

Reproducibility:

Nearest neighbor matching involves a random component only when `m.order = "random"` (or when the propensity is estimated using a method with randomness; see `distance` for details), so a seed must be set in that case using `set.seed()` to ensure reproducibility. Otherwise, it is purely deterministic, and any ties are broken based on the order in which the data appear.

Outputs

All outputs described in `matchit()` are returned with `method = "nearest"`. When `replace = TRUE`, the `subclass` component is omitted. `include.obj` is ignored.

References

In a manuscript, you don't need to cite another package when using `method = "nearest"` because the matching is performed completely within *MatchIt*. For example, a sentence might read:

Nearest neighbor matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R.

See Also

`matchit()` for a detailed explanation of the inputs and outputs of a call to `matchit()`.

`method_optimal()` for optimal pair matching, which is similar to nearest neighbor matching without replacement except that an overall distance criterion is minimized (i.e., as an alternative to specifying `m.order`).

Examples

```
data("lalonde")

# 1:1 greedy NN matching on the PS
m.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "nearest")
m.out1
summary(m.out1)

# 3:1 NN Mahalanobis distance matching with
# replacement within a PS caliper
m.out2 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "nearest",
                  replace = TRUE,
                  mahvars = ~ age + educ + re74 + re75,
                  ratio = 3,
                  caliper = .02)
m.out2
summary(m.out2, un = FALSE)

# 1:1 NN Mahalanobis distance matching within calipers
# on re74 and re75 and exact matching on married and race
m.out3 <- matchit(treat ~ age + educ + re74 + re75,
                  data = lalonde,
                  method = "nearest",
                  distance = "mahalanobis",
                  exact = ~ married + race,
                  caliper = c(re74 = .2, re75 = .15))
m.out3
summary(m.out3, un = FALSE)

# 2:1 variable ratio NN matching on the PS
m.out4 <- matchit(treat ~ age + educ + race + nodegree +
```

```
                    married + re74 + re75,
                  data = lalonde,
                  method = "nearest",
                  ratio = 2,
                  min.controls = 1,
                  max.controls = 12)
m.out4
summary(m.out4, un = FALSE)

# Some units received 1 match and some received 12
table(table(m.out4$subclass[m.out4$treat == 0]))
```

`method_optimal` *Optimal Pair Matching*

Description

In `matchit()`, setting `method = "optimal"` performs optimal pair matching. The matching is optimal in the sense that that sum of the absolute pairwise distances in the matched sample is as small as possible. The method functionally relies on `optmatch::fullmatch()`.

Advantages of optimal pair matching include that the matching order is not required to be specified and it is less likely that extreme within-pair distances will be large, unlike with nearest neighbor matching. Generally, however, as a subset selection method, optimal pair matching tends to perform similarly to nearest neighbor matching in that similar subsets of units will be selected to be matched.

This page details the allowable arguments with `method = "optmatch"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for optimal pair matching:

```
matchit(formula,
        data = NULL,
        method = "optimal",
        distance = "glm",
        link = "logit",
        distance.options = list(),
        estimand = "ATT",
        exact = NULL,
        mahvars = NULL,
        antiexact = NULL,
        discard = "none",
        reestimate = FALSE,
        s.weights = NULL,
        ratio = 1,
        min.controls = NULL,
        max.controls = NULL,
        verbose = FALSE,
        ...)
```

Arguments

formula	a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the matching. This formula will be supplied to the functions that estimate the distance measure.
data	a data frame containing the variables named in formula. If not found in data, the variables will be sought in the environment.
method	set here to "optimal".
distance	the distance measure to be used. See distance for allowable options. Can be supplied as a distance matrix.
link	when distance is specified as a method of estimating propensity scores, an additional argument controlling the link function used in estimating the distance measure. See distance for allowable options with each option.
distance.options	a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to distance.
estimand	a string containing the desired estimand. Allowable options include "ATT" and "ATC". See Details.
exact	for which variables exact matching should take place.
mahvars	for which variables Mahalanobis distance matching should take place when distance corresponds to a propensity score (e.g., for caliper matching or to discard units for common support). If specified, the distance measure will not be used in matching.
antiexact	for which variables anti-exact matching should take place. Anti-exact matching is processed using optmatch::antiExactMatch().
discard	a string containing a method for discarding units outside a region of common support. Only allowed when distance is not "mahalanobis" and not a matrix.
reestimate	if discard is not "none", whether to re-estimate the propensity score in the remaining sample prior to matching.
s.weights	the variable containing sampling weights to be incorporated into propensity score models and balance statistics.
ratio	how many control units should be matched to each treated unit for k:1 matching. For variable ratio matching, see section "Variable Ratio Matching" in Details below.
min.controls, max.controls	for variable ratio matching, the minimum and maximum number of controls units to be matched to each treated unit. See section "Variable Ratio Matching" in Details below.
verbose	logical; whether information about the matching process should be printed to the console. What is printed depends on the matching method. Default is FALSE for no printing other than warnings.
...	additional arguments passed to optmatch::fullmatch(). Allowed arguments include tol and solver. See the optmatch::fullmatch() documentation for details. In general, tol should be set to a low number (e.g., 1e-7) to get a more precise solution (default is 1e-3). The arguments replace, caliper, and m.order are ignored with a warning.

Details

Mahalanobis Distance Matching:

Mahalanobis distance matching can be done one of two ways:

1. If no propensity score needs to be estimated, `distance` should be set to `"mahalanobis"`, and Mahalanobis distance matching will occur using all the variables in `formula`. Arguments to `discard` and `mahvars` will be ignored. For example, to perform simple Mahalanobis distance matching, the following could be run:

   ```
   matchit(treat ~ X1 + X2, method = "nearest",
           distance = "mahalanobis")
   ```

   With this code, the Mahalanobis distance is computed using `X1` and `X2`, and matching occurs on this distance. The `distance` component of the `matchit()` output will be empty.

2. If a propensity score needs to be estimated for common support with `discard`, `distance` should be whatever method is used to estimate the propensity score or a vector of distance measures, i.e., it should not be `"mahalanobis"`. Use `mahvars` to specify the variables used to create the Mahalanobis distance. For example, to perform Mahalanobis after discarding units outside the common support of the propensity score in both groups, the following could be run:

   ```
   matchit(treat ~ X1 + X2 + X3, method = "nearest",
           distance = "glm", discard = "both",
           mahvars = ~ X1 + X2)
   ```

   With this code, `X1`, `X2`, and `X3` are used to estimate the propensity score (using the `"glm"` method, which by default is logistic regression), which is used to identify the common support. The actual matching occurs on the Mahalanobis distance computed only using `X1` and `X2`, which are supplied to `mahvars`. The estimated propensity scores will be included in the `distance` component of the `matchit()` output.

Estimand:

The `estimand` argument controls whether control units are selected to be matched with treated units (`estimand = "ATT"`) or treated units are selected to be matched with control units (`estimand = "ATC"`). The "focal" group (e.g., the treated units for the ATT) is typically made to be the smaller treatment group, and a warning will be thrown if it is not set that. Setting `estimand = "ATC"` is equivalent to swapping all treated and control labels for the treatment variable. When `estimand = "ATC"`, the `match.matrix` component of the output will have the names of the control units as the rownames and be filled with the names of the matched treated units (opposite to when `estimand = "ATT"`). Note that the argument supplied to `estimand` doesn't necessarily correspond to the estimand actually targeted; it is merely a switch to trigger which treatment group is considered "focal".

Variable Ratio Matching:

`matchit()` can perform variable ratio matching, which involves matching a different number of control units to each treated unit. When `ratio` > 1, rather than requiring all treated units to receive `ratio` matches, the arguments to `max.controls` and `min.controls` can be specified to control the maximum and minimum number of matches each treated unit can have. `ratio` controls how many total control units will be matched: `n1 * ratio` control units will be matched, where `n1` is the number of treated units, yielding the same total number of matched controls as fixed ratio matching does.

Variable ratio matching can be used with any `distance` specification. `ratio` does not have to be an integer but must be greater than 1 and less than `n0/n1`, where `n0` and `n1` are the number of control and treated units, respectively. Setting `ratio = n0/n1` performs a restricted form of full matching where all control units are matched. If `min.controls` is not specified, it is set to 1 by default. `min.controls` must be less than `ratio`, and `max.controls` must be greater than `ratio`. See the Examples section of `method_nearest()` for an example of their use, which is the same as it is with optimal matching.

Outputs

All outputs described in `matchit()` are returned with `method = "optimal"`. When `include.obj = TRUE` in the call to `matchit()`, the output of the call to `optmatch::fullmatch()` will be included in the output. When `exact` is specified, this will be a list of such objects, one for each stratum of the `exact` variables.

Note

Optimal pair matching is a restricted form of optimal full matching where the number of treated units in each subclass is equal to 1, whereas in unrestricted full matching, multiple treated units can be assigned to the same subclass. `optmatch::pairmatch()` is simply a wrapper for `optmatch::fullmatch()`, which performs optimal full matching and is the workhorse for `method_full`. In the same way, `matchit()` uses `optmatch::fullmatch()` under the hood, imposing the restrictions that make optimal full matching function like optimal pair matching (which is simply to set `min.controls >= 1` and to pass `ratio` to the `mean.controls` argument). This distinction is not important for regular use but may be of interest to those examining the source code.

The option `"optmatch_max_problem_size"` is automatically set to `Inf` during the matching process, different from its default in *optmatch*. This enables matching problems of any size to be run, but may also let huge, infeasible problems get through and potentially take a long time or crash R. See `optmatch::setMaxProblemSize()` for more details.

A preprocessing algorithm describe by Sävje (2020; doi:10.1214/19STS739) is used to improve the speed of the matching when 1:1 matching on a propensity score. It does so by adding an additional constraint that guarantees a solution as optimal as the solution that would have been found without the constraint, and that constraint often dramatically reduces the size of the matching problem at no cost. However, this may introduce differences between the results obtained by *MatchIt* and by *optmatch*, though such differences will shrink when smaller values of `tol` are used.

References

In a manuscript, be sure to cite the following paper if using `matchit()` with `method = "optimal"`:

Hansen, B. B., & Klopfer, S. O. (2006). Optimal Full Matching and Related Designs via Network Flows. Journal of Computational and Graphical Statistics, 15(3), 609–627. doi:10.1198/106186006X137047

For example, a sentence might read:

Optimal pair matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R, which calls functions from the optmatch package (Hansen & Klopfer, 2006).

See Also

matchit() for a detailed explanation of the inputs and outputs of a call to matchit().

optmatch::fullmatch(), which is the workhorse.

method_full for optimal full matching, of which optimal pair matching is a special case, and which relies on similar machinery.

Examples

```
data("lalonde")

#1:1 optimal PS matching with exact matching on race
m.out1 <- matchit(treat ~ age + educ + race +
                    nodegree + married + re74 + re75,
                  data = lalonde,
                  method = "optimal",
                  exact = ~race)
m.out1
summary(m.out1)

#2:1 optimal matching on the scaled Euclidean distance
m.out2 <- matchit(treat ~ age + educ + race +
                    nodegree + married + re74 + re75,
                  data = lalonde,
                  method = "optimal",
                  ratio = 2,
                  distance = "scaled_euclidean")
m.out2
summary(m.out2, un = FALSE)
```

---

method_quick *Fast Generalized Full Matching*

Description

In matchit(), setting method = "quick" performs generalized full matching, which is a form of subclassification wherein all units, both treatment and control (i.e., the "full" sample), are assigned to a subclass and receive at least one match. It uses an algorithm that is extremely fast compared to optimal full matching, which is why it is labeled as "quick", at the expense of true optimality. The method is described in Sävje, Higgins, & Sekhon (2021). The method relies on and is a wrapper for quickmatch::quickmatch().

Advantages of generalized full matching include that the matching order is not required to be specified, units do not need to be discarded, and it is less likely that extreme within-subclass distances will be large, unlike with standard subclassification. The primary output of generalized full matching is a set of matching weights that can be applied to the matched sample; in this way, generalized full matching can be seen as a robust alternative to propensity score weighting, robust in the sense

that the propensity score model does not need to be correct to estimate the treatment effect without bias.

This page details the allowable arguments with `method = "quick"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for generalized full matching:

```
matchit(formula,
        data = NULL,
        method = "quick",
        distance = "glm",
        link = "logit",
        distance.options = list(),
        estimand = "ATT",
        exact = NULL,
        mahvars = NULL,
        discard = "none",
        reestimate = FALSE,
        s.weights = NULL,
        caliper = NULL,
        std.caliper = TRUE,
        verbose = FALSE,
        ...)
```

Arguments

`formula`	a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the matching. This formula will be supplied to the functions that estimate the distance measure.
`data`	a data frame containing the variables named in `formula`. If not found in `data`, the variables will be sought in the environment.
`method`	set here to `"quick"`.
`distance`	the distance measure to be used. See `distance` for allowable options. Cannot be supplied as a matrix.
`link`	when `distance` is specified as a method of estimating propensity scores, an additional argument controlling the link function used in estimating the distance measure. See `distance` for allowable options with each option.
`distance.options`	a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to `distance`.
`estimand`	a string containing the desired estimand. Allowable options include `"ATT"`, `"ATC"`, and `"ATE"`. The estimand controls how the weights are computed; see the Computing Weights section at `matchit()` for details.
`exact`	for which variables exact matching should take place.
`mahvars`	for which variables Mahalanobis distance matching should take place when `distance` corresponds to a propensity score (e.g., to discard units for common support). If specified, the distance measure will not be used in matching.

discard	a string containing a method for discarding units outside a region of common support. Only allowed when distance corresponds to a propensity score.
reestimate	if discard is not "none", whether to re-estimate the propensity score in the remaining sample prior to matching.
s.weights	the variable containing sampling weights to be incorporated into propensity score models and balance statistics.
caliper	the width of the caliper used for caliper matching. A caliper can only be placed on the propensity score and cannot be negative.
std.caliper	logical; when a caliper is specified, whether it is in standard deviation units (TRUE) or raw units (FALSE).
verbose	logical; whether information about the matching process should be printed to the console.
...	additional arguments passed to quickmatch::quickmatch(). Allowed arguments include treatment_constraints, size_constraint, target, and other arguments passed to scclust::sc_clustering() (see quickmatch::quickmatch() for details). In particular, changing seed_method from its default can improve performance. No arguments will be passed to distances::distances(). The arguments replace, ratio, min.controls, max.controls, m.order, and antiexact are ignored with a warning.

Details

Generalized full matching is similar to optimal full matching, but has some additional flexibility that can be controlled by some of the extra arguments available. By default, method = "quick" performs a standard full match in which all units are matched (unless restricted by the caliper) and assigned to a subclass. Each subclass could contain multiple units from each treatment group. The subclasses are chosen to minimize the largest within-subclass distance between units (including between units of the same treatment group). Notably, generalized full matching requires less memory and can run much faster than optimal full matching and optimal pair matching and, in some cases, even than nearest neighbor matching, and it can be used with huge datasets (e.g., in the millions) while running in under a minute.

Outputs

All outputs described in matchit() are returned with method = "quick" except for match.matrix. This is because matching strata are not indexed by treated units as they are in some other forms of matching. When include.obj = TRUE in the call to matchit(), the output of the call to quickmatch::quickmatch() will be included in the output. When exact is specified, this will be a list of such objects, one for each stratum of the exact variables.

References

In a manuscript, be sure to cite the *quickmatch* package if using matchit() with method = "quick". A citation can be generated using citation("quickmatch").

For example, a sentence might read:

Generalized full matching was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R, which calls functions from the quickmatch package (Sävje, Sekhon, & Higgins, 2024).

You should also cite the following paper, which develops and describes the method:

Sävje, F., Higgins, M. J., & Sekhon, J. S. (2021). Generalized Full Matching. *Political Analysis*, 29(4), 423–447. doi:10.1017/pan.2020.32

See Also

`matchit()` for a detailed explanation of the inputs and outputs of a call to `matchit()`.

`quickmatch::quickmatch()`, which is the workhorse.

`method_full` for optimal full matching, which is nearly the same but offers more customizability and more optimal solutions at the cost of speed.

Examples

```
data("lalonde")

# Generalized full PS matching
m.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "quick")
m.out1
summary(m.out1)
```

`method_subclass` *Subclassification*

Description

In `matchit()`, setting `method = "subclass"` performs subclassification on the distance measure (i.e., propensity score). Treatment and control units are placed into subclasses based on quantiles of the propensity score in the treated group, in the control group, or overall, depending on the desired estimand. Weights are computed based on the proportion of treated units in each subclass. Subclassification implemented here does not rely on any other package.

This page details the allowable arguments with `method = "subclass"`. See `matchit()` for an explanation of what each argument means in a general context and how it can be specified.

Below is how `matchit()` is used for subclassification:

```
matchit(formula,
        data = NULL,
        method = "subclass",
        distance = "glm",
        link = "logit",
        distance.options = list(),
        estimand = "ATT",
        discard = "none",
```

```
          reestimate = FALSE,
          s.weights = NULL,
          verbose = FALSE,
          ...)
```

Arguments

formula	a two-sided formula object containing the treatment and covariates to be used in creating the distance measure used in the subclassification.
data	a data frame containing the variables named in formula. If not found in data, the variables will be sought in the environment.
method	set here to "subclass".
distance	the distance measure to be used. See distance for allowable options. Must be a vector of distance scores or the name of a method of estimating propensity scores.
link	when distance is specified as a string, an additional argument controlling the link function used in estimating the distance measure. See distance for allowable options with each option.
distance.options	a named list containing additional arguments supplied to the function that estimates the distance measure as determined by the argument to distance.
estimand	the target estimand. If "ATT", the default, subclasses are formed based on quantiles of the distance measure in the treated group; if "ATC", subclasses are formed based on quantiles of the distance measure in the control group; if "ATE", subclasses are formed based on quantiles of the distance measure in the full sample. The estimand also controls how the subclassification weights are computed; see the Computing Weights section at matchit() for details.
discard	a string containing a method for discarding units outside a region of common support.
reestimate	if discard is not "none", whether to re-estimate the propensity score in the remaining sample prior to subclassification.
s.weights	the variable containing sampling weights to be incorporated into propensity score models and balance statistics.
verbose	logical; whether information about the matching process should be printed to the console.
...	additional arguments that control the subclassification: subclass either the number of subclasses desired or a vector of quantiles used to divide the distance measure into subclasses. Default is 6. min.n the minimum number of units of each treatment group that are to be assigned each subclass. If the distance measure is divided in such a way that fewer than min.n units of a treatment group are assigned a given subclass, units from other subclasses will be reassigned to fill the deficient subclass. Default is 1. The arguments exact, mahvars, replace, m.order, caliper (and related arguments), and ratio are ignored with a warning.

Details

After subclassification, effect estimates can be computed separately in the subclasses and combined, or a single marginal effect can be estimated by using the weights in the full sample. When using the weights, the method is sometimes referred to as marginal mean weighting through stratification (MMWS; Hong, 2010) or fine stratification weighting (Desai et al., 2017). The weights can be interpreted just like inverse probability weights. See `vignette("estimating-effects")` for details.

Changing `min.n` can change the quality of the weights. Generally, a low `min.w` will yield better balance because subclasses only contain units with relatively similar distance values, but may yield higher variance because extreme weights can occur due to there being few members of a treatment group in some subclasses. When `min.n = 0`, some subclasses may fail to contain units from both treatment groups, in which case all units in such subclasses will be dropped.

Note that subclassification weights can also be estimated using *WeightIt*, which provides some additional methods for estimating propensity scores. Where propensity score-estimation methods overlap, both packages will yield the same weights.

Outputs

All outputs described in `matchit()` are returned with `method = "subclass"` except that `match.matrix` is excluded and one additional component, `q.cut`, is included, containing a vector of the distance measure cutpoints used to define the subclasses. Note that when `min.n > 0`, the subclass assignments may not strictly obey the quantiles listed in `q.cut`. `include.obj` is ignored.

References

In a manuscript, you don't need to cite another package when using `method = "subclass"` because the subclassification is performed completely within *MatchIt*. For example, a sentence might read:

Propensity score subclassification was performed using the MatchIt package (Ho, Imai, King, & Stuart, 2011) in R.

It may be a good idea to cite Hong (2010) or Desai et al. (2017) if the treatment effect is estimated using the subclassification weights.

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

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

See Also

`matchit()` for a detailed explanation of the inputs and outputs of a call to `matchit()`.

`method_full` for optimal full matching and `method_quick` for generalized full matching, which are similar to subclassification except that the number of subclasses and subclass membership are chosen to optimize the within-subclass distance.

Examples

```
data("lalonde")

# PS subclassification for the ATT with 7 subclasses
s.out1 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "subclass",
                  subclass = 7)
s.out1
summary(s.out1, subclass = TRUE)

# PS subclassification for the ATE with 10 subclasses
# and at least 2 units in each group per subclass
s.out2 <- matchit(treat ~ age + educ + race + nodegree +
                    married + re74 + re75,
                  data = lalonde,
                  method = "subclass",
                  subclass = 10,
                  estimand = "ATE",
                  min.n = 2)
s.out2
summary(s.out2)
```

`plot.matchit` *Generate Balance Plots after Matching and Subclassification*

Description

Generates plots displaying distributional balance and overlap on covariates and propensity scores before and after matching and subclassification. For displaying balance solely on covariate standardized mean differences, see `plot.summary.matchit()`. The plots here can be used to assess to what degree covariate and propensity score distributions are balanced and how weighting and discarding affect the distribution of propensity scores.

Usage

```
## S3 method for class 'matchit'
plot(x, type = "qq", interactive = TRUE, which.xs = NULL, data = NULL, ...)

## S3 method for class 'matchit.subclass'
plot(x, type = "qq", interactive = TRUE, which.xs = NULL, subclass, ...)
```

Arguments

`x`	a `matchit` object; the output of a call to `matchit()`.
`type`	the type of plot to display. Options include `"qq"`, `"ecdf"`, `"density"`, `"jitter"`, and `"histogram"`. See Details. Default is `"qq"`. Abbreviations allowed.

interactive	logical; whether the graphs should be displayed in an interactive way. Only applies for type = "qq", "ecdf", "density", and "jitter". See Details.
which.xs	with type = "qq", "ecdf", or "density", for which covariate(s) plots should be displayed. Factor variables should be named by the original variable name rather than the names of individual dummy variables created after expansion with model.matrix. Can be supplied as a character vector or a one-sided formula.
data	an optional data frame containing variables named in which.xs but not present in the matchit object.
...	arguments passed to plot() to control the appearance of the plot. Not all options are accepted.
subclass	with subclassification and type = "qq", "ecdf", or "density", whether to display balance for individual subclasses, and, if so, for which ones. Can be TRUE (display plots for all subclasses), FALSE (display plots only in aggregate), or the indices (e.g., 1:6) of the specific subclasses for which to display balance. When unspecified, if interactive = TRUE, you will be asked for which subclasses plots are desired, and otherwise, plots will be displayed only in aggregate.

Details

plot.matchit() makes one of five different plots depending on the argument supplied to type. The first three, "qq", "ecdf", and "density", assess balance on the covariates. When interactive = TRUE, plots for three variables will be displayed at a time, and the prompt in the console allows you to move on to the next set of variables. When interactive = FALSE, multiple pages are plotted at the same time, but only the last few variables will be visible in the displayed plot. To see only a few specific variables at a time, use the which.xs argument to display plots for just those variables. If fewer than three variables are available (after expanding factors into their dummies), interactive is ignored.

With type = "qq", empirical quantile-quantile (eQQ) plots are created for each covariate before and after matching. The plots involve interpolating points in the smaller group based on the weighted quantiles of the other group. When points are approximately on the 45-degree line, the distributions in the treatment and control groups are approximately equal. Major deviations indicate departures from distributional balance. With variable with fewer than 5 unique values, points are jittered to more easily visualize counts.

With type = "ecdf", empirical cumulative distribution function (eCDF) plots are created for each covariate before and after matching. Two eCDF lines are produced in each plot: a gray one for control units and a black one for treated units. Each point on the lines corresponds to the proportion of units (or proportionate share of weights) less than or equal to the corresponding covariate value (on the x-axis). Deviations between the lines on the same plot indicates distributional imbalance between the treatment groups for the covariate. The eCDF and eQQ statistics in summary.matchit() correspond to these plots: the eCDF max (also known as the Kolmogorov-Smirnov statistic) and mean are the largest and average vertical distance between the lines, and the eQQ max and mean are the largest and average horizontal distance between the lines.

With type = "density", density plots are created for each covariate before and after matching. Two densities are produced in each plot: a gray one for control units and a black one for treated units. The x-axis corresponds to the value of the covariate and the y-axis corresponds to the density or probability of that covariate value in the corresponding group. For binary covariates, bar plots

are produced, having the same interpretation. Deviations between the black and gray lines represent imbalances in the covariate distribution; when the lines coincide (i.e., when only the black line is visible), the distributions are identical.

The last two plots, `"jitter"` and `"histogram"`, visualize the distance (i.e., propensity score) distributions. These plots are more for heuristic purposes since the purpose of matching is to achieve balance on the covariates themselves, not the propensity score.

With `type = "jitter"`, a jitter plot is displayed for distance values before and after matching. This method requires a distance variable (e.g., a propensity score) to have been estimated or supplied in the call to `matchit()`. The plot displays individuals values for matched and unmatched treatment and control units arranged horizontally by their propensity scores. Points are jitter so counts are easier to see. The size of the points increases when they receive higher weights. When `interactive = TRUE`, you can click on points in the graph to identify their rownames and indices to further probe extreme values, for example. With subclassification, vertical lines representing the subclass boundaries are overlay on the plots.

With `type = "histogram"`, a histogram of distance values is displayed for the treatment and control groups before and after matching. This method requires a distance variable (e.g., a propensity score) to have been estimated or supplied in the call to `matchit()`. With subclassification, vertical lines representing the subclass boundaries are overlay on the plots.

With all methods, sampling weights are incorporated into the weights if present.

Note

Sometimes, bugs in the plotting functions can cause strange layout or size issues. Running `frame()` or `dev.off()` can be used to reset the plotting pane (note the latter will delete any plots in the plot history).

See Also

`summary.matchit()` for numerical summaries of balance, including those that rely on the eQQ and eCDF plots.

`plot.summary.matchit()` for plotting standardized mean differences in a Love plot.

`cobalt::bal.plot()` for displaying distributional balance in several other ways that are more easily customizable and produce *ggplot2* objects. *cobalt* functions natively support `matchit` objects.

Examples

```
data("lalonde")

m.out <- matchit(treat ~ age + educ + married +
                   race + re74,
                 data = lalonde,
                 method = "nearest")
plot(m.out, type = "qq",
     interactive = FALSE,
     which.xs = ~age + educ + re74)
plot(m.out, type = "histogram")

s.out <- matchit(treat ~ age + educ + married +
```

```
                  race + nodegree + re74 + re75,
                data = lalonde,
                method = "subclass")
plot(s.out, type = "density",
     interactive = FALSE,
     which.xs = ~age + educ + re74,
     subclass = 3)
plot(s.out, type = "jitter",
     interactive = FALSE)
```

`plot.summary.matchit` *Generate a Love Plot of Standardized Mean Differences*

Description

Generates a Love plot, which is a dot plot with variable names on the y-axis and standardized mean differences on the x-axis. Each point represents the standardized mean difference of the corresponding covariate in the matched or unmatched sample. Love plots are a simple way to display covariate balance before and after matching. The plots are generated using `dotchart()` and `points()`.

Usage

```
## S3 method for class 'summary.matchit'
plot(
  x,
  abs = TRUE,
  var.order = "data",
  threshold = c(0.1, 0.05),
  position = "bottomright",
  ...
)
```

Arguments

`x`	a `summary.matchit` object; the output of a call to `summary.matchit()`. The `standardize` argument must be set to `TRUE` (which is the default) in the call to `summary`.
`abs`	`logical`; whether the standardized mean differences should be displayed in absolute value (`TRUE`, default) or not `FALSE`.
`var.order`	how the variables should be ordered. Allowable options include `"data"`, ordering the variables as they appear in the `summary` output; `"unmatched"`, ordered the variables based on their standardized mean differences before matching; `"matched"`, ordered the variables based on their standardized mean differences after matching; and `"alphabetical"`, ordering the variables alphabetically. Default is `"data"`. Abbreviations allowed.

`threshold` numeric values at which to place vertical lines indicating a balance threshold. These can make it easier to see for which variables balance has been achieved given a threshold. Multiple values can be supplied to add multiple lines. When `abs = FALSE`, the lines will be displayed on both sides of zero. The lines are drawn with `abline` with the linetype (`lty`) argument corresponding to the order of the entered variables (see options at `par()`). The default is `c(.1, .05)` for a solid line (`lty = 1`) at .1 and a dashed line (`lty = 2`) at .05, indicating acceptable and good balance, respectively. Enter a value as `NA` to skip that value of `lty` (e.g., `c(NA, .05)` to have only a dashed vertical line at .05).

`position` the position of the legend. Should be one of the allowed keyword options supplied to `x` in `legend()` (e.g., `"right"`, `"bottomright"`, etc.). Default is `"bottomright"`. Set to `NULL` for no legend to be included. Note that the legend will cover up points if you are not careful; setting `var.order` appropriately can help in avoiding this.

`...` ignored.

Details

For matching methods other than subclassification, `plot.summary.matchit` uses `x$sum.all[,"Std. Mean Diff."]` and `x$sum.matched[,"Std. Mean Diff."]` as the x-axis values. For subclassification, in addition to points for the unadjusted and aggregate subclass balance, numerals representing balance in individual subclasses are plotted if `subclass = TRUE` in the call to `summary`. Aggregate subclass standardized mean differences are taken from `x$sum.across[,"Std. Mean Diff."]` and the subclass-specific mean differences are taken from `x$sum.subclass`.

Value

A plot is displayed, and `x` is invisibly returned.

Author(s)

Noah Greifer

See Also

`summary.matchit()`, `dotchart()`

`cobalt::love.plot()` is a more flexible and sophisticated function to make Love plots and is also natively compatible with `matchit` objects.

Examples

```
data("lalonde")
m.out <- matchit(treat ~ age + educ + married +
                   race + re74,
                 data = lalonde,
                 method = "nearest")
plot(summary(m.out, interactions = TRUE),
     var.order = "unmatched")
```

```
s.out <- matchit(treat ~ age + educ + married +
                   race + nodegree + re74 + re75,
                 data = lalonde,
                 method = "subclass")
plot(summary(s.out, subclass = TRUE),
     var.order = "unmatched",
     abs = FALSE)
```

`rbind.matchdata` *Append matched datasets together*

Description

These functions are `rbind()` methods for objects resulting from calls to `match_data()` and `get_matches()`. They function nearly identically to `rbind.data.frame()`; see Details for how they differ.

Usage

```
## S3 method for class 'matchdata'
rbind(..., deparse.level = 1)

## S3 method for class 'getmatches'
rbind(..., deparse.level = 1)
```

Arguments

`...`	Two or more `matchdata` or `getmatches` objects the output of calls to `match_data()` and `get_matches()`, respectively. Supplied objects must either be all `matchdata` objects or all `getmatches` objects.
`deparse.level`	Passed to `rbind()`.

Details

`rbind()` appends two or more datasets row-wise. This can be useful when matching was performed separately on subsets of the original data and they are to be combined into a single dataset for effect estimation. Using the regular `data.frame` method for `rbind()` would pose a problem, however; the `subclass` variable would have repeated names across different datasets, even though units only belong to the subclasses in their respective datasets. `rbind.matchdata()` renames the subclasses so that the correct subclass membership is maintained.

The supplied matched datasets must be generated from the same original dataset, that is, having the same variables in it. The added components (e.g., weights, subclass) can be named differently in different datasets but will be changed to have the same name in the output.

`rbind.getmatches()` and `rbind.matchdata()` are identical.

Value

An object of the same class as those supplied to it (i.e., a `matchdata` object if `matchdata` objects are supplied and a `getmatches` object if `getmatches` objects are supplied). `rbind()` is called on the objects after adjusting the variables so that the appropriate method will be dispatched corresponding to the class of the original data object.

Author(s)

Noah Greifer

See Also

`match_data()`, `rbind()`

See `vignettes("estimating-effects")` for details on using `rbind()` for effect estimation after subsetting the data.

Examples

```
data("lalonde")

# Matching based on race subsets
m.out_b <- matchit(treat ~ age + educ + married +
                     nodegree + re74 + re75,
                   data = subset(lalonde, race == "black"))
md_b <- match_data(m.out_b)

m.out_h <- matchit(treat ~ age + educ + married +
                     nodegree + re74 + re75,
                   data = subset(lalonde, race == "hispan"))
md_h <- match_data(m.out_h)

m.out_w <- matchit(treat ~ age + educ + married +
                     nodegree + re74 + re75,
                   data = subset(lalonde, race == "white"))
md_w <- match_data(m.out_w)

#Bind the datasets together
md_all <- rbind(md_b, md_h, md_w)

#Subclass conflicts are avoided
levels(md_all$subclass)
```

`summary.matchit` *View a balance summary of a* `matchit` *object*

Description

Computes and prints balance statistics for `matchit` and `matchit.subclass` objects. Balance should be assessed to ensure the matching or subclassification was effective at eliminating treatment group imbalance and should be reported in the write-up of the results of the analysis.

Usage

```
## S3 method for class 'matchit'
summary(
  object,
  interactions = FALSE,
  addlvariables = NULL,
  standardize = TRUE,
  data = NULL,
  pair.dist = TRUE,
  un = TRUE,
  improvement = FALSE,
  ...
)

## S3 method for class 'matchit.subclass'
summary(
  object,
  interactions = FALSE,
  addlvariables = NULL,
  standardize = TRUE,
  data = NULL,
  pair.dist = FALSE,
  subclass = FALSE,
  un = TRUE,
  improvement = FALSE,
  ...
)

## S3 method for class 'summary.matchit'
print(x, digits = max(3, getOption("digits") - 3), ...)
```

Arguments

`object`	a `matchit` object; the output of a call to `matchit()`.
`interactions`	`logical`; whether to compute balance statistics for two-way interactions and squares of covariates. Default is `FALSE`.
`addlvariables`	additional variable for which balance statistics are to be computed along with the covariates in the `matchit` object. Can be entered in one of three ways: as a data frame of covariates with as many rows as there were units in the original `matchit()` call, as a string containing the names of variables in `data`, or as a right-sided `formula` with the additional variables (and possibly their

transformations) found in data, the environment, or the matchit object. Balance on squares and interactions of the additional variables will be included if interactions = TRUE.

standardize — logical; whether to compute standardized (TRUE) or unstandardized (FALSE) statistics. The standardized statistics are the standardized mean difference and the mean and maximum of the difference in the (weighted) empirical cumulative distribution functions (ECDFs). The unstandardized statistics are the raw mean difference and the mean and maximum of the quantile-quantile (QQ) difference. Variance ratios are produced either way. See Details below. Default is TRUE.

data — a optional data frame containing variables named in addlvariables if specified as a string or formula.

pair.dist — logical; whether to compute average absolute pair distances. For matching methods that don't include a match.matrix component in the output (i.e., exact matching, coarsened exact matching, full matching, and subclassification), computing pair differences can take a long time, especially for large datasets and with many covariates. For other methods (i.e., nearest neighbor, optimal, and genetic matching), computation is fairly quick. Default is FALSE for subclassification and TRUE otherwise.

un — logical; whether to compute balance statistics for the unmatched sample. Default TRUE; set to FALSE for more concise output.

improvement — logical; whether to compute the percent reduction in imbalance. Default FALSE. Ignored if un = FALSE.

... — ignored.

subclass — after subclassification, whether to display balance for individual subclasses, and, if so, for which ones. Can be TRUE (display balance for all subclasses), FALSE (display balance only in aggregate), or the indices (e.g., 1:6) of the specific subclasses for which to display balance. When anything other than FALSE, aggregate balance statistics will not be displayed. Default is FALSE.

x — a summay.matchit or summary.matchit.subclass object; the output of a call to summary().

digits — the number of digits to round balance statistics to.

Details

summary() computes a balance summary of a matchit object. This include balance before and after matching or subclassification, as well as the percent improvement in balance. The variables for which balance statistics are computed are those included in the formula, exact, and mahvars arguments to matchit(), as well as the distance measure if distance is was supplied as a numeric vector or method of estimating propensity scores. The X component of the matchit object is used to supply the covariates.

The standardized mean differences are computed both before and after matching or subclassification as the difference in treatment group means divided by a standardization factor computed in the unmatched (original) sample. The standardization factor depends on the argument supplied to estimand in matchit(): for "ATT", it is the standard deviation in the treated group; for "ATC", it is the standard deviation in the control group; for "ATE", it is the square root of the average of

the variances within each treatment group. The post-matching mean difference is computed with weighted means in the treatment groups using the matching or subclassification weights.

The variance ratio is computed as the ratio of the treatment group variances. Variance ratios are not computed for binary variables because their variance is a function solely of their mean. After matching, weighted variances are computed using the formula used in `cov.wt()`. The percent reduction in bias is computed using the log of the variance ratios.

The eCDF difference statistics are computed by creating a (weighted) eCDF for each group and taking the difference between them for each covariate value. The eCDF is a function that outputs the (weighted) proportion of units with covariate values at or lower than the input value. The maximum eCDF difference is the same thing as the Kolmogorov-Smirnov statistic. The values are bounded at zero and one, with values closer to zero indicating good overlap between the covariate distributions in the treated and control groups. For binary variables, all eCDF differences are equal to the (weighted) difference in proportion and are computed that way.

The QQ difference statistics are computed by creating two samples of the same size by interpolating the values of the larger one. The values are arranged in order for each sample. The QQ difference for each quantile is the difference between the observed covariate values at that quantile between the two groups. The difference is on the scale of the original covariate. Values close to zero indicate good overlap between the covariate distributions in the treated and control groups. A weighted interpolation is used for post-matching QQ differences. For binary variables, all QQ differences are equal to the (weighted) difference in proportion and are computed that way.

The pair distance is the average of the absolute differences of a variable between pairs. For example, if a treated unit was paired with four control units, that set of units would contribute four absolute differences to the average. Within a subclass, each combination of treated and control unit forms a pair that contributes once to the average. The pair distance is described in Stuart and Green (2008) and is the value that is minimized when using optimal (full) matching. When `standardize = TRUE`, the standardized versions of the variables are used, where the standardization factor is as described above for the standardized mean differences. Pair distances are not computed in the unmatched sample (because there are no pairs). Because pair distance can take a while to compute, especially with large datasets or for many covariates, setting `pair.dist = FALSE` is one way to speed up `summary()`.

The effective sample size (ESS) is a measure of the size of a hypothetical unweighted sample with roughly the same precision as a weighted sample. When non-uniform matching weights are computed (e.g., as a result of full matching, matching with replacement, or subclassification), the ESS can be used to quantify the potential precision remaining in the matched sample. The ESS will always be less than or equal to the matched sample size, reflecting the loss in precision due to using the weights. With non-uniform weights, it is printed in the sample size table; otherwise, it is removed because it does not contain additional information above the matched sample size.

After subclassification, the aggregate balance statistics are computed using the subclassification weights rather than averaging across subclasses.

All balance statistics (except pair differences) are computed incorporating the sampling weights supplied to `matchit()`, if any. The unadjusted balance statistics include the sampling weights and the adjusted balance statistics use the matching weights multiplied by the sampling weights.

When printing, `NA` values are replaced with periods (`.`), and the pair distance column in the unmatched and percent balance improvement components of the output are omitted.

Value

For `matchit` objects, a `summary.matchit` object, which is a list with the following components:

`call`	the original call to `matchit()`
`nn`	a matrix of the sample sizes in the original (unmatched) and matched samples
`sum.all`	if `un = TRUE`, a matrix of balance statistics for each covariate in the original (unmatched) sample
`sum.matched`	a matrix of balance statistics for each covariate in the matched sample
`reduction`	if `improvement = TRUE`, a matrix of the percent reduction in imbalance for each covariate in the matched sample

For `match.subclass` objects, a `summary.matchit.subclass` object, which is a list as above containing the following components:

`call`	the original call to `matchit()`
`sum.all`	if `un = TRUE`, a matrix of balance statistics for each covariate in the original sample
`sum.subclass`	if `subclass` is not `FALSE`, a list of matrices of balance statistics for each subclass
`sum.across`	a matrix of balance statistics for each covariate computed using the subclassification weights
`reduction`	if `improvement = TRUE`, a matrix of the percent reduction in imbalance for each covariate in the matched sample
`qn`	a matrix of sample sizes within each subclass
`nn`	a matrix of the sample sizes in the original (unmatched) and matched samples

See Also

`summary()` for the generic method; `plot.summary.matchit()` for making a Love plot from `summary()` output.

`cobalt::bal.tab.matchit()`, which also displays balance for `matchit` objects.

Examples

```
data("lalonde")
m.out <- matchit(treat ~ age + educ + married +
                   race + re74,
                 data = lalonde,
                 method = "nearest",
                 exact = ~ married,
                 replace = TRUE)

summary(m.out, interactions = TRUE)

s.out <- matchit(treat ~ age + educ + married +
                   race + nodegree + re74 + re75,
                 data = lalonde,
                 method = "subclass")
```

```
summary(s.out, addlvariables = ~log(age) + I(re74==0))

summary(s.out, subclass = TRUE)
```

Index