Package 'miceExt'

March 6, 2018

Title Extension Package to 'mice'

Version 1.1.0

Maintainer Tobias Schumacher <tobias.schumacher1@rwth-aachen.de>

Description Extends and builds on the 'mice' package by
adding a functionality to perform multivariate predictive mean matching on
imputed data as well as new functionalities to perform predictive mean
matching on factor variables.

Depends R (>= 3.3), mice (>= 2.46.0)

Imports RANN (>= 2.5.1), RANN.L1 (>= 2.5)

License GPL-2

Author Tobias Schumacher [aut, cre],
Philipp Gaffert [aut],
Stef van Buuren [ctb],
Karin Groothuis-Oudshoorn [ctb]

URL http://github.com/tobiasschumacher/miceExt

BugReports http://github.com/tobiasschumacher/miceExt/issues

Encoding UTF-8

LazyData true

RoxygenNote 6.0.1

Suggests testthat

NeedsCompilation no

Repository CRAN

Date/Publication 2018-03-05 23:46:37 UTC

R topics documented:

`boys_data` *Growth of Dutch boys*

Description

Modified sample of the `boys` data set that is included in `mice`.

Format

A dataset with 100 observations of the following 9 variables:

age Decimal age (0-21 years)

hgt Height (cm)

wgt Weight (kg)

bmi Body mass index

hc Head circumference (cm)

gen Genital Tanner stage (G1-G5)

phb Pubic hair (Tanner P1-P6)

tv Testicular volume (ml)

reg Region (north, east, west, south, city)

Details

The original data set `boys` that is included in `mice` is a random sample of 10% from the cross-sectional data used to construct the Dutch growth references 1997, and contains information height, weight, head circumference and puberty of 748 Dutch boys. From this data, a sample of 100 rows has been taken and modified such that in each row, the values in the columns `hc`, `gen` and `phb` are either blockwise `NA` or blockwise non-`NA`.

Source

Fredriks, A.M,, van Buuren, S., Burgmeijer, R.J., Meulmeester JF, Beuker, R.J., Brugman, E., Roede, M.J., Verloove-Vanhorick, S.P., Wit, J.M. (2000) Continuing positive secular growth change in The Netherlands 1955-1997. *Pediatric Research*, **47**, 316-323.

Fredriks, A.M., van Buuren, S., Wit, J.M., Verloove-Vanhorick, S.P. (2000). Body index measurements in 1996-7 compared with 1980. *Archives of Disease in Childhood*, **82**, 107-112.-734. '@keywords datasets

See Also

`boys`

mammal_data *Mammal sleep data*

Description

Modified version of the mammal sleep data set that is included in mice.

Format

A dataset with 62 observations of the following 11 variables:

species Species of animal

bw Body weight (kg)

brw Brain weight (g)

sws Slow wave ("nondreaming") sleep (hrs/day)

ps Paradoxical ("dreaming") sleep (hrs/day)

ts Total sleep (hrs/day) (sum of slow wave and paradoxical sleep)

mls Maximum life span (years)

gt Gestation time (days)

pi Predation index (1-5), 1 = least likely to be preyed upon

sei Sleep exposure index (1-5), 1 = least exposed (e.g. animal sleeps in a well-protected den), 5 = most exposed

odi Overall danger index (1-5) based on the above two indices and other information, 1 = least danger (from other animals), 5 = most danger (from other animals)

Details

The original dataset was from a study by Allison and Cicchetti (1976) of 62 mammal species on the interrelationship between sleep, ecological, and constitutional variables, and was adapted in the mice-package. It contains missing values on five variables and has been modified such that for each row, the entries in the column tuples (sws,ps) and (mls,gt) are either pairwiseNA or pairwise non-NA.

Source

Allison, T., Cicchetti, D.V. (1976). Sleep in Mammals: Ecological and Constitutional Correlates. Science, 194(4266), 732-734.

See Also

mammalsleep

`mice.binarize` *Binarize Factor Columns in Data Frames*

Description

This function replaces factor columns in data frames in-place by a set of binary columns which represent the so-called one-hot encoding of this factor. More precisely, a column of a factor with `n` levels will be transformed into a set of `n` binary columns, each representing exactly one category of the original factor. Hence, the value `1` occurs in a column if and only if the original factor had the value corresponding to that column.
Further, this function also returns a weights vector and a predictor matrix that fit to the binarized data frame. The weights vector is recommended to be used as input for `mice.post.matching()`, as this avoids the effect of overweighing a big set of binary columns relating to one factor against other columns within one imputation block, while the predictor matrix, when used as input parameter in `mice()`, ensures that binary columns that relate to the same factor do not predict each other.

Usage

```
mice.binarize(data, include_ordered = TRUE, include_observed = FALSE,
  cols = NULL, blocks = NULL, weights = rep(1, ncol(data)),
  pred_matrix = (1 - diag(1, ncol(data))))
```

Arguments

`data`	Matrix or data frame that contains factor columns which we want to convert into an equivalent set of binary columns.
`include_ordered`	Logical variable indicating whether we also want to transform ordered factors. Default is `TRUE`.
`include_observed`	Logical variable indicating whether we also want to transform factor columns in which all values are observed. Default is `FALSE`.
`cols`	Numerical vector corresponding to the indices or character vector corresponding to the names of factor columns which we want to transform. By default, its value is `NULL`, indicating that the algorithm automatically identifies all factor columns that are to be binarized. If however the user specifies its value, the function exclusively transforms the specified columns, ignoring the values of the previous optional parameters.
`blocks`	Optional vector or list of vectors specifying the column tuples that are to be imputed on blockwise later when running `mice.post.matching()`. If this parameter is specified, the function will dectect all factor columns withing this argument and binarize these, while also producing a corresponding expanded block vector that can be found in the output of this function. In particular, whenever this parameter is specified, the values of the previous three parameters will be ignored. For a detailed explanation about the valid formats of this parameter, see section ***input formats*** below.

The default of this parameter is `blocks = NULL`, in which case the output parameter `blocks` will also be `NULL`.

`weights` Optional numeric vector or list of numeric vectors that allocates weights to the columns in the data, in particular on those that are going to be imputed on blockwise. If specified, the input will be transformed into vector format (if necessary), and columns that correspond to factor columns that are binarized will also be expanded. Within this expansion, all weights that belong to factor columns are thereby also divided by the number of levels of their corresponding factor column. This is done to ensure a balanced weighting in the later matching process within `mice.post.matching()`, as in the case that a factor column with many levels is in the same block with factor columns that much fewer levels or even with single numeric columns, the predictive means of the dummy columns of this factor would have much more impact on the matching than predictive means of the other column(s) simply because the latter would be outnumbered.
The default of this parameter is `weights = rep(1, ncol(data))`, which initially assigns the weight `1` to all columns before expanding them and reducing the weight of binarized columns as explained above. In any way, it is strongly recommended to use the transformed output `weights` as the `weights` parameter in `mice.post.matching()`.
Note that specifying this parameter in list format is only allowed if blocks have been specified.

`pred_matrix` A custom predictor matrix relating to input `data`, which will get transformed into the format that fits to the binarized output data frame. The result of this transformation will be stored in the `pred_matrix` element of the output and should then be used as the `predictorMatrix` parameter in `mice()` to ensure that binary columns relating to the same factor column in the original data do not predict each other, yielding cleaner imputation models. If not specified, the default is the massive imputation predictor matrix.

Value

List containing the following five elements:

`data` The binarized data frame.

`par_list` A list containing the original data frame as well as some parameters with further information on the transformation. This list is needed to retransform the (possibly imputed) data at later stager via the `mice.factorize()` function, and should not be edited by the user under any circumstance. Next to the original data, the most notable element of this list would be "dummy_cols", which itself is a list of the column tuples that correspond to the transformed factor column from the original data set, and therefore works perfectly as input for `mice.post.matching()`.

`blocks` If input parameter `blocks` has been specified, an expanded version of that input is returned in vector format via this element. It should be used as input parameter `blocks` in `mice.post.matching()` later, after imputing the binarized data via `mice()` first.

`weights` Transformed version of input parameter `weights` in vector format that should be used as input parameter `weights` in `mice.post.matching()` later, after imputing the binarized data via `mice()` first. If input parameter `weights` has not been specified, a default vector is still going to be output.

`pred_matrix` Transformed version of input `pred_matrix` that should be used as the input argument `predictorMatrix` of `mice()`.

Input Formats

Within `mice.binarize()` and `mice.post.matching()`, there are two formats that can be used to specify the input parameters `blocks` and `weights`, namely the *list format* and the *vector format*. The basic idea behind the list format is that we exclusively specify parameters for those column blocks that we want to impute on and summarize them in a list where each element is a vector that represents one such block, while in the vector format we use a single vector in which each element represents one column in the data, and therefore also specify information for columns that we do not want to impute on. The exact use of these two formats on both the `blocks` and the `weights` parameter will be illustrated in the following.

`blocks` **1. List Format**

To specify the imputation `blocks` using the list format, a list of atomic vectors has to be passed to the `blocks` parameter, and each vector in this list represents a column block that has to be imputed on. These vectors can either contain the names of the columns in this block as character strings or the corresponding column indices in numerical format. Note that this input list must not contain any duplicate columns among and within its elements. If there is only a single block to impute on, a single atomic vector representing this tuple may also be passed to `blocks`.

Example:

Within this and the following examples of this section, we consider the `boys_data` data set which contains 9 columns, out of which the column tuples `(hgt, bmi)` and `(hc, gen, phb)` have identical missing value patterns, while the columns `gen` and `phb` are also categorical. (check `?boys_data` for further details on the data).

If we now wanted to specify these blocks in list format, we would have to write

```
blocks = list(c("hgt","bmi"), c("hc", "gen", "phb"))
```

or analogously, when using column indices instead,

```
blocks = list(c(2,4), c(5,6,7)).
```

2. Vector Format

If we want to specify the imputation `blocks` via the vector format, a single vector with as many elements as there are columns in the data has to be used. Each element of this vector assigns a block number to its corresponding column, and columns that have the same number are imputed together, while columns that are not to be imputed have to carry the value `0`. All block numbers have to be integral, starting from `1`, and the total number of imputation blocks has to be the maximum block number.

Example:

Again we want to blockwise impute the column tuples `(hgt, bmi)` and `(hc, gen, phb)` from `boys_data`. To specify these blocks via the vector notation, we assign block number `1` to the columns of the first tuple and group number `2` to the columns of the second tuple, while all other columns have block number `0`. Hence, we would pass

```
blocks = c(0,1,0,1,2,2,2,0,0)
```

to `mice.post.matching()`.

`weights` **1. List Format**

To specify the imputation `weights` using the list format, the corresponding imputations blocks must have been specified in list format as well. In this case, the `weights` list has to be the

same length as blocks, and each of its elements has a numeric vector of the same length as its corresponding column block, thereby assigning each of its columns a (strictly postitive) weight. If we do not want to apply weights to a single tuple, we can write a single `0`, `1` or `NULL` in the corresponding spot of the list.

Example:
In our example, we want to assign the `hgt` column a `1.5` times higher weight than `bmi`, while not assigning any values to the second tuple at all. To achieve that, we specify
`weights = list(c(3,2), NULL)`.

2. Vector Format
When specifying the imputation `weights` via the vector format, once again a single vector with as many elements as there are columns in the data has to be used, in which each element assigns a weight to its corresponding column. Weights of columns that are not imputed on will have no effect, while in all blocks that weights should not be applied on, each column should carry the same value. In general, the value `1` should be used as the standard weight value for all columns that either are not imputed on or that weights are not applied on.

Example:
We want to assign the same weights to the first tuple as in the previous example, while not assigning weights the second block again. In this case, we would use the vector
`weights = c(1,3,1,2,1,1,1,1,1)`
in which all the values of the unimputed and unweighted columns are `1`.

Internally, `mice.post.matching()` converts both parameters into the vector format as this works best within the main iteration over all columns of the data. Hence, the output parameters `blocks` and `weights` are also in list format.

Author(s)

Tobias Schumacher, Philipp Gaffert

See Also

`mice.factorize`, `mice.post.matching`, `mice`

Examples

```
#------------------------------------------------------------------------------
# first set of examples illustrating basic functionality
#------------------------------------------------------------------------------

# binarize all factor columns in boys_data that contain NAs
boys_bin <- mice.binarize(boys_data)

# binarize only column 'gen' in boys_data
boys_bin <- mice.binarize(boys_data, cols = c("gen"))

# binarize all factor columns with the blocks ('hgt','bmi') and ('gen','phb')
# to impute on these (binarized) blocks later
boys_bin <- mice.binarize(boys_data, blocks = list(c("hgt","bmi"), c("gen", "phb")))
```

```
# read out binarized data
boys_bin$data



## Not run:
#------------------------------------------------------------------------------
# Examples illustrating the combined usage of blocks and weights, relating to
# the examples in the input format section above. As before, we want to impute
# on the column tuples ('hgt','bmi') and ('hc','gen','phb) from boys_data, while
# assigning weights to the first block, in which 'hgt' gets a 1.5 times
# higher weight than 'bmi'. The second tuple is not weighted.
#------------------------------------------------------------------------------

## Now there are four options to specify the blocks and weights:

# First option: specify blocks and weights in list format
boys_bin <- mice.binarize(data = boys_data,
                              blocks = list(c("hgt","bmi"), c("hc","gen","phb")),
                              weights = list(c(3,2), NULL))

# or equivalently, using colums indices:
boys_bin <- mice.binarize(data = boys_data,
                          blocks = list(c(2,4), c(5,6,7)),
                          weights = list(c(3,2), NULL))

# Second option: specify blocks in list and weights in vector format
post_mammal <- mice.binarize(data = boys_data,
                                 blocks = c(0,1,0,1,2,2,2,0,0),
                                 weights = c(1,3,1,2,1,1,1,1,1))

# Third option: specify blocks in list format and weights in vector format
post_mammal <- mice.binarize(data = boys_data,
                                 blocks = list(c("hgt","bmi"), c("hc","gen", "phb")),
                                 weights = c(1,3,1,2,1,1,1,1,1))

# Fourth option: specify blocks in vector format and weights in list format.
# Note that the block number determines which tuple in the weights list it
# corresponds to, and within each tuple in the list the weight correspondence is
# determinded by left to right order of the data columns
post_mammal <- mice.binarize(data = boys_data,
                                 blocks = c(0,1,0,1,2,2,2,0,0),
                                 weights = list(c(3,2), NULL))

# check expanded blocks vector
boys_bin$blocks

# check expanded weights vector
boys_bin$weights
```

```
#-------------------------------------------------------------------------------
# Example that illustrates the combined functionalities of mice.binarize(),
# mice.factorize() and mice.post.matching() on the data set 'boys_data', which
# contains the column blocks ('hgt','bmi') and ('hc','gen','phb') that have
# identical missing value patterns, and out of which the columns 'gen' and
# 'phb' are factors. We are going to impute both tuples blockwise, while
# binarizing the factor columns first. Note that we never need to specify any
# blocks or columns to binarize, as these are all determined automatically
#-------------------------------------------------------------------------------

# By default, mice.binarize() expands all factor columns that contain NAs,
# so the columns 'gen' and 'phb' are automatically binarized
boys_bin <- mice.binarize(boys_data)

# Run mice on binarized data, note that we need to use boys_bin$data to grab
# the actual binarized data and that we use the output predictor matrix
# boys_bin$pred_matrix which is recommended for obtaining better imputation
# models
mids_boys <- mice(boys_bin$data, predictorMatrix = boys_bin$pred_matrix)

# It is very likely that mice imputed multiple ones among one set of dummy
# variables, so we need to post-process. As recommended, we also use the output
# weights from mice.binarize(), which yield a more balanced weighting on the
# column tuple ('hc','gen','phb') within the matching. As in previous examples,
# both tuples are automatically discovered and imputed on
post_boys <- mice.post.matching(mids_boys, weights = boys_bin$weights)

# Now we can safely retransform to the original data, with non-binarized
# imputations
res_boys <- mice.factorize(post_boys$midsobj, boys_bin$par_list)

# Analyze the distribution of imputed variables, e.g. of the column 'gen',
# using the mice version of with()
with(res_boys, table(gen))



#-------------------------------------------------------------------------------
# Similar example to the previous, that also works on 'boys_data' and
# illustrates some more advanced funtionalities of all three functions in miceExt:
# This time we only want to post-process the column block ('gen','phb'), while
# weighting the first of these tuples twice as much as the second. Within the
# matching, we want to avoid matrix computations by using the euclidian distance
# to determine the donor pool, and we want to draw from three donors only.
#-------------------------------------------------------------------------------

# Binarize first, we specify blocks in list format with a single block, so we
# can omit an enclosing list. Similarly, we also specify weights in list format.
# Both blocks and weights will be expanded and can be accessed from the output
# to use them in mice.post.matching() later
boys_bin <- mice.binarize(boys_data,
                          blocks = c("gen", "phb"),
                          weights = c(2,1))
```

```
# Run mice on binarized data, again use the output predictor matrix from
# mice.binarize()
mids_boys <- mice(boys_bin$data, predictorMatrix = boys_bin$pred_matrix)

# Post-process the binarized columns. We use blocks and weights from the previous
# output, and set 'distmetric' and 'donors' as announced in the example
# description
post_boys <- mice.post.matching(mids_boys,
                                blocks = boys_bin$blocks,
                                weights = boys_bin$weights,
                                distmetric = "euclidian",
                                donors = 3L)

# Finally, we can retransform to the original format
res_boys <- mice.factorize(post_boys$midsobj, boys_bin$par_list)

## End(Not run)
```

`mice.factorize` *Transform Imputations of Binarized Data Into Their Corresponding Factors*

Description

This function acts as the counterpart to `mice.binarize`, as it effectively retransforms imputations of binarized data that `mice` has been run on and that has been post-processed via `mice.post.matching` after. The post-processing is usually necessary as `mice` is very likely to impute multiple ones among the dummy columns belonging to to a single factor entry. The resulting `mice::mids` object is not suited for further `mice.mids()` iterations or the use of `plot`, but works well as input to `with()`.

Usage

```
mice.factorize(obj, par_list)
```

Arguments

`obj`	`mice::mids` object resulting from a call of `mice.post.matching()` and whose underlying data frame results from a call of `mice::binarize()`.
`par_list`	List that has been returned in a previous call of `mice::binarize()` next to the underlying data of the argument `obj`.

Value

A `mice::mids` object in which data and imputations have been retransformed from their respective binarized versions in the input `obj`. As this isn't a proper result of a mice iteration and many of the attributes of `obj` cannot be transformed well, only the slots `data`, `nmis`, `where` and `imp`, which are needed in `with()`, are not `NULL`. In particular, it would not work as input for `mice.mids()`.

Author(s)

Tobias Schumacher, Philipp Gaffert

See Also

mice.binarize, mice.post.matching, mice

Examples

```
## Not run:
#-------------------------------------------------------------------------------
# Example that illustrates the combined functionalities of mice.binarize(),
# mice.factorize() and mice.post.matching() on the data set 'boys_data', which
# contains the column blocks ('hgt','bmi') and ('hc','gen','phb') that have
# identical missing value patterns, and out of which the columns 'gen' and
# 'phb' are factors. We are going to impute both tuples blockwise, while
# binarizing the factor columns first. Note that we never need to specify any
# blocks or columns to binarize, as these are all determined automatically
#-------------------------------------------------------------------------------

# By default, mice.binarize() expands all factor columns that contain NAs,
# so the columns 'gen' and 'phb' are automatically binarized
boys_bin <- mice.binarize(boys_data)

# Run mice on binarized data, note that we need to use boys_bin$data to grab
# the actual binarized data and that we use the output predictor matrix
# boys_bin$pred_matrix which is recommended for obtaining better imputation
# models
mids_boys <- mice(boys_bin$data, predictorMatrix = boys_bin$pred_matrix)

# It is very likely that mice imputed multiple ones among one set of dummy
# variables, so we need to post-process. As recommended, we also use the output
# weights from mice.binarize(), which yield a more balanced weighting on the
# column tuple ('hc','gen','phb') within the matching. As in previous examples,
# both tuples are automatically discovered and imputed on
post_boys <- mice.post.matching(mids_boys, weights = boys_bin$weights)

# Now we can safely retransform to the original data, with non-binarized
# imputations
res_boys <- mice.factorize(post_boys$midsobj, boys_bin$par_list)

# Analyze the distribution of imputed variables, e.g. of the column 'gen',
# using the mice version of with()
with(res_boys, table(gen))



#-------------------------------------------------------------------------------
# Similar example to the previous, that also works on 'boys_data' and
# illustrates some more advanced funtionalities of all three functions in miceExt:
# This time we only want to post-process the column block ('gen','phb'), while
```

```
# weighting the first of these tuples twice as much as the second. Within the
# matching, we want to avoid matrix computations by using the euclidian distance
# to determine the donor pool, and we want to draw from three donors only.
#------------------------------------------------------------------------------

# Binarize first, we specify blocks in list format with a single block, so we
# can omit an enclosing list. Similarly, we also specify weights in list format.
# Both blocks and weights will be expanded and can be accessed from the output
# to use them in mice.post.matching() later
boys_bin <- mice.binarize(boys_data,
                          blocks = c("gen", "phb"),
                          weights = c(2,1))

# Run mice on binarized data, again use the output predictor matrix from
# mice.binarize()
mids_boys <- mice(boys_bin$data, predictorMatrix = boys_bin$pred_matrix)

# Post-process the binarized columns. We use blocks and weights from the previous
# output, and set 'distmetric' and 'donors' as announced in the example
# description
post_boys <- mice.post.matching(mids_boys,
                                blocks = boys_bin$blocks,
                                weights = boys_bin$weights,
                                distmetric = "euclidian",
                                donors = 3L)

# Finally, we can retransform to the original format
res_boys <- mice.factorize(post_boys$midsobj, boys_bin$par_list)

## End(Not run)
```

`mice.post.matching` *Post-processing of Imputed Data by Multivariate Predictive Mean Matching*

Description

Performs multivariate predictive mean matching (PMM) on a set of columns that have been imputed on by the functionalities of the `mice` package. Also offers a functionality to match imputations against observed variables.

Usage

```
mice.post.matching(obj, blocks = NULL, donors = 5L, weights = NULL,
  distmetric = "residual", matchtype = 1L, match_vars = NULL,
  ridge = 1e-05, minvar = 1e-04, maxcor = 0.99)
```

Arguments

obj	mice::mids object that has been returned by a previous run of mice() or mice.mids() and whose imputations we want to post-process.
blocks	Vector or list of vectors specifying the column tuples that are to be imputed on blockwise. For a detailed explanation about the valid formats of this parameter, see section ***input formats*** below. Each element of each specified block has to be included in the visitSequence of the given mids object, and the imputation method that was applied on each of these columns has to be either pmm or norm. Univariate blocks are also allowed if they have been imputed on via mice.impute.norm(), or if we want to match them against an observed variable that is specified in the parameter match_vars. The default is blocks = NULL, which tells this function to automatically determine and impute on column blocks that have identical missing value patterns.
donors	Integer or integer vector indicating the size of the donor pool among which a draw in the matching step is made. If only a single number of donors is specified, it is applied on all blocks, otherwise this parameter needs to be a vector with as many elements as there blocks, specifying for each of these blocks how many donors are to be drawn from. The default is donors = 5L. Setting donors = 1L always selects the closest match (nearest neighbor), but is not recommended.
weights	Numeric vector or list of numeric vectors that allocates weights to the columns of each block, giving us the possibility to punish or mitigate differences in certain columns of the data when computing the distances in the matching step to determine the donor pool. Further details on the valid formats of this parameter can be found below in section ***input formats***. The default is weights = NULL, meaning that no weights should be applied to any column at all. Note that in order to avoid any ambiguities, specifying this parameter in list format is only allowed if blocks have been explicitly specified and NOT automatically determined.
distmetric	Character string or character vector that determines which mathematical metric we want to use to compute the distances between the multivariate y_obs and y_mis. If only a single metric is specified, it is applied on all blocks, otherwise this parameter needs to be a vector with as many elements as there blocks, specifying for each of these blocks which metric there is to use. The options are "euclidian", "manhattan", "mahalanobis" and "residual". The first three options refer to the distance metrics of the same name, the latter is a variant of the mahalanobis distance in which we consider the residual covariance cov(y_hat_obs - y_obs) of the predicted model. This distance has been proposed and recommended by Little (1988) and is also the default. Note that the first two options are faster though, as they do not involve any matrix computations.
matchtype	Integer of integer vector indicating the type of matching distance to be used in PMM. If only a single matchtype is specified, it is applied on all blocks, otherwise this parameter needs to be a vector with as many elements as there blocks, specifying for each of these blocks which matchtype has to be used.

The default choice (`matchtype = 1L`) calculates the distance between the *predicted* values of `y_obs` and the *drawn* values of `y_mis` (called type-1 matching). Other choices are `matchtype = 0L` distance between predicted values) and `matchtype = 2L` (distance between drawn values).

`match_vars` Vector specifying for each tuple which additional variable has to be matched against. Can be an integer or character vector, either specifying column indices or column names. `match_vars` must be the same length as `blocks` with `0`-elements or `""`-elements to disable this functionality for certain groups. Default is `match_vars = NULL`, which completely disables this functionality.

`ridge` The ridge penalty used in an internal call of `mice::.norm.draw()` to prevent problems with multicollinearity. Can be a single number or a numeric vector. If only a single ridge value is specified, it is applied on all blocks, otherwise this parameter needs to be a vector with as many elements as there blocks, specifying for each of these blocks which ridge value has to be used.
The default is `ridge = 1e-05`, which means that 0.01 percent of the diagonal is added to the cross-product. Larger ridges may result in more biased estimates. For highly noisy data (e.g. many junk variables), set `ridge = 1e-06` or even lower to reduce bias. For highly collinear data, set `ridge = 1e-04` or higher.

`minvar` The minimum variance that we require predictors to have when building a linear model to compute the `y_hat`-values. Can be a single number or a numeric vector. If only a single value is specified, it is applied on all blocks, otherwise this parameter needs to be a vector with as many elements as there blocks, specifying for each of these blocks which minimum variance will b allowed. The default is `minvar = 1e-04`.

`maxcor` The maximum correlation that we allow predictors to have when building a linear model to compute the `y_hat`-values. Can be a single number or a numeric vector. If only a single value is specified, it is applied on all blocks, otherwise this parameter needs to be a vector with as many elements as there blocks, specifying for each of these blocks which maximum variance will be allowed. The default is `maxcor = 0.99`.

Value

List containing the following two elements:

`midsobj` `mice::mids` object that differs from the input object only in the imputations that have been post-processed, and the `call` and `loggedEvents` attributes that have been updated. In particular, those post-processed imputations are not affecting the `chainMean` or `chainVar`-attributes, and hence, `plot()` will not consider them either.

`blocks` Set of column blocks in list format that multivariate imputation has been performed on. It is equivalent to the input parameter `blocks` if it has been specified by the user, otherwise those column tuples have been determined internally.

Input Formats

Within `mice.post.matching()` and `mice.binarize()`, there are two formats that can be used to specify the input parameters `blocks` and `weights`, namely the *list format* and the *vector format*. The basic idea behind the list format is that we exclusively specify parameters for those column

blocks that we want to impute on and summarize them in a list where each element is a vector that represents one such block, while in the vector format we use a single vector in which each element represents one column in the data, and therefore also specify information for columns that we do not want to impute on. The exact use of these two formats on both the `blocks` and the `weights` parameter will be illustrated in the following.

`blocks` **1. List Format**

To specify the imputation `blocks` using the list format, a list of atomic vectors has to be passed to the `blocks` parameter, and each vector in this list represents a column block that has to be imputed on. These vectors can either contain the names of the columns in this block as character strings or the corresponding column indices in numerical format. Note that this input list must not contain any duplicate columns among and within its elements. If there is only a single block to impute on, a single atomic vector representing this tuple may also be passed to `blocks`.

Example:

Within this and the following examples of this section, we consider the `mammal_data` data set which contains 11 columns, out of which the column tuples `(sws, ps)` and `(mls, gt)` have identical missing value patterns (check `?mammal_data` for further details on the data).
If we now wanted to specify these blocks in list format, we would have to write
`blocks = list(c("sws","ps"), c("mls", "gt"))`
or analogously, when using column indices instead,
`blocks = list(c(4,5), c(7,8))`.

2. Vector Format

If we want to specify the imputation `blocks` via the vector format, a single vector with as many elements as there are columns in the data has to be used. Each element of this vector assigns a block number to its corresponding column, and columns that have the same number are imputed together, while columns that are not to be imputed have to carry the value `0`. All block numbers have to be integral, starting from `1`, and the total number of imputation blocks has to be the maximum block number.

Example:

Again we want to blockwise impute the column tuples `(sws, ps)` and `(mls, gt)` from `mammal_data`. To specify these blocks via the vector notation, we assign block number `1` to the columns of the first tuple and group number `2` to the columns of the second tuple, while all other columns have block number `0`. Hence, we would pass
`blocks = c(0,0,0,1,1,0,2,2,0,0,0)`
to `mice.post.matching()`.

`weights` **1. List Format**

To specify the imputation `weights` using the list format, the corresponding imputations blocks must have been specified in list format as well. In this case, the `weights` list has to be the same length as blocks, and each of its elements has a numeric vector of the same length as its corresponding column block, thereby assigning each of its columns a (strictly postitive) weight. If we do not want to apply weights to a single tuple, we can write a single `0`, `1` or `NULL` in the corresponding spot of the list.

Example:

In our example, we want to assign the `sws` column a `1.5` times higher weight than `ps`, while not assigning any values to the second tuple at all. To achieve that, we specify
`weights = list(c(3,2), NULL)`.

2. Vector Format
When specifying the imputation `weights` via the vector format, once again a single vector with as many elements as there are columns in the data has to be used, in which each element assigns a weight to its corresponding column. Weights of columns that are not imputed on will have no effect, while in all blocks that weights should not be applied on, each column should carry the same value. In general, the value `1` should be used as the standard weight value for all columns that either are not imputed on or that weights are not applied on.

Example:
We want to assign the same weights to the first tuple as in the previous example, while not assigning weights the second block again. In this case, we would use the vector
`weights = c(1,1,1,3,2,1,1,1,1,1,1)`
in which all the values of the unimputed and unweighted columns are `1`.

Internally, `mice.post.matching()` converts both parameters into the list format as this is more natural to iterate on. Hence, the output `blocks` parameter also is in list format.

Algorithmic Details

The algorithm basically iterates over the `m` imputations of the input `mice::mids` object and each column tuple in `blocks`, each time following these two main steps:

Prediction First, we iterate over the columns of the current block, collecting the complete column y on which we want to impute, along with the matrix x of its predictors. Among the values of eqny, we identify the observed values y_{obs} and the missing values `y_{mis}` along with their corresponding predictors x_{obs} and `x_{mis}`, restricting ourselves to only those values whose predictors are complete, i.e. don't contain any missing values themselves. Note that if the sets of predictor variables strongly vary among all the columns in the current tuple, it may occur that there are no common predictors in which case the function breaks.
From the observed values and their predictors, we build a Bayesian linear regression model and, depending on the specified matching type, use the model's coefficients and drawn regression weights β to compute the predicted means $\hat{y}_{obs}$ and $\hat{y}_{mis}$ which we then save in a matrix $\hat{Y}$.

Matching After iterating over the columns in the current tuple and building the matrix $\hat{Y}$, we match the multivariate predictions of the missing values in $\hat{Y}$ against the predictions of the observed values. More precisely, for each $\hat{y}_{mis}$, we perform a k-nearest-neighbor search among all values $\hat{y}_{obs}$, where k is the number of donors specified by the user, and then randomly sample one element of those k nearest neighbors which is then used to impute the missing value tuple in y. The distance metric that is used to determine the nearest neighbors is specified by the user as well, and in case of Euclidian or Manhattan distance its computation is very straightforward. If the Mahalanobis distance or the residual distance (as proposed by Little) has been selected, we first compute the corresponding covariance matrix and its (pseudo) inverse via its eigen decomposition, and then use it to transform the values $\hat{y}_{mis}$ and $\hat{y}_{obs}$ that are fed into the nearest neighbor search. If weights have been specified as well, they are also applied before the kNN-search.
If an additional observed variable to match against has been specified via `match_vars`, the set of predictors and missing values are partitioned by the values of the external variable first, and then the matching is performed within each pair of corresponding subsets of that partition.

Note that the imputed values are only stored as a result and, other than in the `mice` algorithm, are not used to compute predictive means for any other missing value. We exclusively use the

imputed values that are provided within the input mids object.

Author(s)

Tobias Schumacher, Philipp Gaffert

References

Little, R.J.A. (1988), Missing data adjustments in large surveys (with discussion), Journal of Business Economics and Statistics, 6, 287–301.

Van Buuren, S. (2012). Flexible Imputation of Missing Data. CRC/Chapman \& Hall, Boca Raton, FL.

Van Buuren, S., Groothuis-Oudshoorn, K. (2011). mice: Multivariate Imputation by Chained Equations in R. *Journal of Statistical Software*, **45**(3), 1-67. http://www.jstatsoft.org/v45/i03/

See Also

mice, mids-class, mice.binarize, mice.factorize

Examples

```
## Not run:
#-------------------------------------------------------------------------------
# Example on modified 'mammalsleep' data set from mice, that has identical
# missing data patterns on the column tuples ('ps','sws') and ('mls','gt')
#-------------------------------------------------------------------------------

# run mice on data set 'mammal_data' and obtain a mids object to post-process
mids_mammal <- mice(mammal_data)


# run function, as blocks have not been specified, it will automatically detect
# the column tuples with identical missing data patterns and then impute on
# these
post_mammal <- mice.post.matching(mids_mammal)

# read out which column tuples have been imputed on
post_mammal$blocks

# look into imputations within resulting mice::mids object
post_mammal$midsobj$imp



#-------------------------------------------------------------------------------
# Example on original 'mammalsleep' data set from mice, in which we
# want to post-process the imputations in column 'sws' by only imputing values
# from rows whose value in 'pi' matches the value of 'pi' in the row we impute
# on.
```

```
#------------------------------------------------------------------------------

# run mice on data set 'mammal_data' and obtain a mids object to post-process
mids_mammal <- mice(mammalsleep)


# run function, specify 'sws' as the column to impute on, and specify 'pi' as
# the observed variable to consider in the matching
post_mammal <- mice.post.matching(mids_mammal, blocks = "sws", match_vars = "pi")


# look into imputations within resulting mice::mids object
post_mammal$midsobj$imp



#------------------------------------------------------------------------------
# Examples illustrating the combined usage of blocks and weights, relating to
# the examples in the input format section above. As before, we want to impute
# on the column tuples ('ps','sws') and ('mls','gt') from mammal_data, while
# this time assigning weights to the first block, in which 'ps' gets a 1.5 times
# higher weight than 'sws'. The second tuple is not weighted.
#------------------------------------------------------------------------------

# run mice() first
mids_mammal <- mice(mammal_data)

## Now there are five options to specify the blocks and weights:

# First option: specify blocks and weights in list format
post_mammal <- mice.post.matching(obj = mids_mammal,
                                  blocks = list(c("sws","ps"), c("mls","gt")),
                                  weights = list(c(3,2), NULL))

# or equivalently, using colums indices:
post_mammal <- mice.post.matching(obj = mids_mammal,
                                  blocks = list(c(4,5), c(7,8)),
                                  weights = list(c(3,2), NULL))

# Second option: specify blocks and weights in vector format
post_mammal <- mice.post.matching(obj = mids_mammal,
                                  blocks = c(0,0,0,1,1,0,2,2,0,0,0),
                                  weights = c(1,1,1,3,2,1,1,1,1,1,1))

# Third option: specify blocks in list format and weights in vector format
post_mammal <- mice.post.matching(obj = mids_mammal,
                                  blocks = list(c("sws","ps"), c("mls","gt")),
                                  weights = c(1,1,1,3,2,1,1,1,1,1,1))

# Fourth option: specify blocks in vector format and weights in list format.
# Note that the block number determines which tuple in the weights list it
# corresponds to, and within each tuple in the list the weight correspondence is
# determinded by left to right order of the data columns
```

```
post_mammal <- mice.post.matching(obj = mids_mammal,
                                  blocks = c(0,0,0,1,1,0,2,2,0,0,0),
                                  weights = list(c(3,2), NULL))

# Fifth option: only specify weights in vector format. If the user knows
# beforehand that at least the column tuple he wants to impute and use weights
# on have the same missing value patterns, he can assign weights to these using
# the vector format, while letting mice.post.matching() find all other blocks
# with identical missing value patterns - possibly even more than just
# ('ps','sws') and ('mls','gt')
post_mammal <- mice.post.matching(obj = mids_mammal,
                                  weights = c(1,1,1,3,2,1,1,1,1,1,1))



#------------------------------------------------------------------------------
# Example that illustrates the combined functionalities of mice.binarize(),
# mice.factorize() and mice.post.matching() on the data set 'boys_data', which
# contains the column blocks ('hgt','bmi') and ('hc','gen','phb') that have
# identical missing value patterns, and out of which the columns 'gen' and
# 'phb' are factors. We are going to impute both tuples blockwise, while
# binarizing the factor columns first. Note that we never need to specify any
# blocks or columns to binarize, as these are all determined automatically
#------------------------------------------------------------------------------

# By default, mice.binarize() expands all factor columns that contain NAs,
# so the columns 'gen' and 'phb' are automatically binarized
boys_bin <- mice.binarize(boys_data)

# Run mice on binarized data, note that we need to use boys_bin$data to grab
# the actual binarized data and that we use the output predictor matrix
# boys_bin$pred_matrix which is recommended for obtaining better imputation
# models
mids_boys <- mice(boys_bin$data, predictorMatrix = boys_bin$pred_matrix)

# It is very likely that mice imputed multiple ones among one set of dummy
# variables, so we need to post-process. As recommended, we also use the output
# weights from mice.binarize(), which yield a more balanced weighting on the
# column tuple ('hc','gen','phb') within the matching. As in previous examples,
# both tuples are automatically discovered and imputed on
post_boys <- mice.post.matching(mids_boys, weights = boys_bin$weights)

# Now we can safely retransform to the original data, with non-binarized
# imputations
res_boys <- mice.factorize(post_boys$midsobj, boys_bin$par_list)

# Analyze the distribution of imputed variables, e.g. of the column 'gen',
# using the mice version of with()
with(res_boys, table(gen))



#------------------------------------------------------------------------------
```

```
# Similar example to the previous, that also works on 'boys_data' and
# illustrates some more advanced funtionalities of all three functions in miceExt:
# This time we only want to post-process the column block ('gen','phb'), while
# weighting the first of these tuples twice as much as the second. Within the
# matching, we want to avoid matrix computations by using the euclidian distance
# to determine the donor pool, and we want to draw from three donors only.
#------------------------------------------------------------------------------

# Binarize first, we specify blocks in list format with a single block, so we
# can omit an enclosing list. Similarly, we also specify weights in list format.
# Both blocks and weights will be expanded and can be accessed from the output
# to use them in mice.post.matching() later
boys_bin <- mice.binarize(boys_data,
                          blocks = c("gen", "phb"),
                          weights = c(2,1))

# Run mice on binarized data, again use the output predictor matrix from
# mice.binarize()
mids_boys <- mice(boys_bin$data, predictorMatrix = boys_bin$pred_matrix)

# Post-process the binarized columns. We use blocks and weights from the previous
# output, and set 'distmetric' and 'donors' as announced in the example
# description
post_boys <- mice.post.matching(mids_boys,
                                blocks = boys_bin$blocks,
                                weights = boys_bin$weights,
                                distmetric = "euclidian",
                                donors = 3L)

# Finally, we can retransform to the original format
res_boys <- mice.factorize(post_boys$midsobj, boys_bin$par_list)

## End(Not run)
```

`miceExt` *miceExt: Extension Package to mice*

Description

This package extends and builds on the `mice` package by adding a functionality to perform multivariate predictive mean matching on imputed data as well as new functionalities to perform predictive mean matching on factor variables.

Details

The `mice` package, which was implemented and published by Stef van Buuren and Karin Groothuis-Oudshoorn in 2001 and has been further developed ever since, is one of most extensive and most

commonly used implementations of multiple imputation within R. Despite its many years of refinement however, there are still some missing data problems that mice does not handle very well, and two of these have now been addressed within the implementation of this package.
First, mice does not provide any option to perform imputation on multiple columns at once, which can, for instance, result in nonsensical output imputations when there are causal relationships between the corresponding attributes, e.g. a 15-year-old person that has a driver's license.
Further, mice still struggles with imputing categorical data, as many internally used imputation methods either are not suited for this kind of data at all or do not necessarily converge to the optimal solution.
Overall, `miceExt` provides three functions, namely

1. `mice.post.matching()`,
2. `mice.binarize()`,
3. `mice.factorize()`,

out of which the first function post-processes results of the `mice()`-algorithm by performing multivariate predictive mean matching on a user-defined set of column tuples, and results in imputations that are always equal to already-observed values, which annihilates the chance of getting unrealistic output values.
The latter two functions tackle the second issue by even extending the functionality of `mice.post.matching()`. The function `mice.binarize()` transforms categorical attributes of a given data frame into a binary dummy representation, which results in an exclusively numerical data set that mice can handle well. Inconsistencies within the imputed dummy columns can then be handled by `mice.post.matching()`, and `mice.factorize()` finally serves the purpose of retransforming the imputed binary data into the corresponding original categories, resulting in a proper imputation of the given categorical data.

Author(s)

Tobias Schumacher, Philipp Gaffert, Stef van Buuren, Karin Groothuis-Oudshoorn

See Also

`mice.post.matching`, `mice.binarize`, `mice.factorize`, `mice`

Index