Package ‘missMethods’

April 1, 2020

Title Methods for Missing Data

Version 0.1.0

Description Supply functions for the creation and handling of missing data as well as tools to evaluate missing data methods. Nearly all possibilities of generating missing data discussed by Santos et. al (2019) <doi:10.1109/ACCESS.2019.2891360> and some additional are implemented. Functions are supplied to compare parameter estimates and imputed values to true values to evaluate missing data methods. Evaluations of these types are done, for example, by Cetin-Berber et al. (2019) <doi:10.1177/0013164418805532> and Kim et al. (2005) <doi:10.1093/bioinformatics/bth499>.

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 7.1.0

Suggests ggplot2, knitr, lpSolve, rmarkdown, testthat (>= 2.1.0), tibble

Imports stats

URL https://github.com/torockel/missMethods

BugReports https://github.com/torockel/missMethods/issues

VignetteBuilder knitr

NeedsCompilation no

Author Tobias Rockel [aut, cre]

Maintainer Tobias Rockel <Rockel.To@gmail.com>

Repository CRAN

Date/Publication 2020-04-01 09:30:02 UTC
**R topics documented:**

- apply_imputation ................................................................. 2
- delete_MAR_1_to_x ............................................................... 4
- delete_MAR_censoring ......................................................... 7
- delete_MAR_one_group ......................................................... 9
- delete_MAR_rank ................................................................. 11
- delete_MCAR ................................................................. 12
- delete_MNAR_1_to_x ............................................................... 13
- delete_MNAR_censoring ......................................................... 15
- delete_MNAR_one_group ......................................................... 16
- delete_MNAR_rank ................................................................. 17
- evaluate_imputation_parameters ............................................. 19
- evaluate_imputed_values ...................................................... 20
- evaluate_parameters ............................................................ 23
- impute_mean ................................................................. 24
- impute_median ................................................................. 25
- impute_mode ................................................................. 26
- impute_sRHD ................................................................. 27
- median.factor ................................................................. 29

**Index** 31

---

**apply_imputation**  
*Apply a function for imputation*

**Description**

Apply a function for imputation over rows, columns or combinations of both

**Usage**

```
apply_imputation(ds, FUN = mean, type = "columnwise", ...)
```

**Arguments**

- `ds`  
a data frame or matrix with missing values
- `FUN`  
the function to be applied for imputation
- `type`  
a string specifying the values used for imputation, see details
- `...`  
进一步 arguments passed to FUN
Details

The functionality of apply_imputation is inspired by the apply function. The function applies a function FUN to impute the missing values in ds. FUN must be a function, which takes a vector as input and returns exactly one value. The argument type is comparable to apply’s MARGIN argument. It specifies the values that are used for the calculation of the imputation values. For example, type = "columnwise" and FUN = mean will impute the mean of the observed values in a column for all missing values in this column. In contrast, type = "rowwise" and FUN = mean will impute the mean of the observed values in a row for all missing values in this row.

List of all implemented types:

- "columnwise" (the default): imputes column by column; all observed values of a column are given to FUN and the returned value is used as the imputation value for all missing values of the column.
- "rowwise": imputes row by row; all observed values of a row are given to FUN and the returned value is used as the imputation value for all missing values of the row.
- "total": All observed values of ds are given to FUN and the returned value is used as the imputation value for all missing values of ds.
- "Winer": The mean value from "columnwise" and "rowwise" is used as the imputation value.
- "Two-way": The sum of the values from "columnwise" and "rowwise" minus "total" is used as the imputation value.

If no value can be given to FUN (for example, if no value in a column is observed and type = "columnwise"), then a warning will be issued and no value will be imputed in the corresponding column or row.

Value

An object of the same class as ds with imputed missing values

A Note for tibble users

If you use tibbles and an error like ‘Lossy cast from ‘value’ <double> to ‘x’ <integer>’ occurs, you will first need to convert all integer columns with missing values to double. Another solution is to convert the tibble with as.data.frame() to a data frame. The data frame will automatically convert integer columns to double columns, if needed.

References


See Also

A convenient interface exists for common cases like mean imputation: impute_mean, impute_median, impute_mode. All these functions call apply_imputation.
Examples

```r
ds <- data.frame(X = 1:20, Y = 101:120)
ds_miss <- delete_MCAR(ds, 0.2)
ds_imp_app <- apply_imputation(ds_miss, FUN = mean, type = "total")
# the same result can be achieved via impute_mean():
ds_imp_mean <- impute_mean(ds_miss, type = "total")
all.equal(ds_imp_app, ds_imp_mean)
```

---

**delete_MAR_1_to_x**  
Create MAR values using MAR1:x

**Description**

Create missing at random (MAR) values using MAR1:x in a data frame or a matrix

**Usage**

```r
delete_MAR_1_to_x(
  ds,
  p,
  miss_cols,
  ctrl_cols,
  x,
  cutoff_fun = median,
  prop = 0.5,
  use_lpSolve = TRUE,
  ordered_as_unordered = FALSE,
  stochastic = FALSE,
  add_realized_x = FALSE,
  ...
)
```

**Arguments**

- **ds**: a data frame or matrix in which missing values will be created
- **p**: a numeric vector with length one or equal to length `miss_cols`; the probability that a value is missing
- **miss_cols**: a vector of column names or indices of columns in which missing values will be created
- **ctrl_cols**: a vector of column names or indices of columns, which controls the creation of missing values in `miss_cols`. Must be of the same length as `miss_cols`.
- **x**: numeric with length one (0 < x < Inf); odds are 1 to x for the probability of a value to be missing in group 1 against the probability of a value to be missing in group 2 (see details)
- **cutoff_fun**: function that calculates the cutoff values in the `ctrl_cols`
delete_MAR_1_to_x

prop numeric of length one; (minimum) proportion of rows in group 1
use_lpSolve logical; should lpSolve be used for the determination of groups, if ctrl_cols[i] is an unordered factor
ordered_as_unordered logical; should ordered factors be treated as unordered factors
stochastic logical; see details
add_realized_x logical; if TRUE the realized odds for miss_cols will be returned (as attribute)
... further arguments passed to cutoff_fun

Details

This function creates missing at random (MAR) values in the columns specified by the argument miss_cols. The probability for missing values is controlled by p. If p is a single number, then the overall probability for a value to be missing will be p in all columns of miss_cols. (Internally p will be replicated to a vector of the same length as miss_cols. So, all p[i] in the following sections will be equal to the given single number p.) Otherwise, p must be of the same length as miss_cols. In this case, the overall probability for a value to be missing will be p[i] in the column miss_cols[i]. The position of the missing values in miss_cols[i] is controlled by ctrl_cols[i].

The following procedure is applied for each pair of ctrl_cols[i] and miss_cols[i] to determine the positions of missing values:

At first, the rows of ds are divided into two groups. Therefore, the cutoff_fun calculates a cutoff value for ctrl_cols[i] (via cutoff_fun(ds[,ctrl_cols[i]],...)). The group 1 consists of the rows, whose values in ctrl_cols[i] are below the calculated cutoff value. If the so defined group 1 is empty, the rows that have a value equal to the cutoff value will be added to this group (otherwise, these rows will belong to group 2). The group 2 consists of the remaining rows, which are not part of group 1. Now the probabilities for the rows in the two groups are set in the way that the odds are 1:x against a missing value in miss_cols[i] for the rows in group 1 compared to the rows in group 2. That means, the probability for a value to be missing in group 1 divided by the probability for a value to be missing in group 2 equals 1 divided by x. For example, for two equal sized groups 1 and 2, ideally the number of observations that are deleted in group 1 compared to the rows in group 2 should equal 1 divided by x. But there are some restrictions, which can lead to some deviations from the odds 1:x (see below).

If stochastic = FALSE (the default), then exactly round(nrow(ds) * p[i]) values will be set NA in column miss_cols[i]. To achieve this, it is possible that the true odds differ from 1:x. The number of observations that are deleted in group 1 and group 2 are chosen to minimize the absolute difference between the realized odds and 1:x. Furthermore, if round(nrow(ds) * p[i]) == 0, then no missing value will be created in miss_cols[i]. If stochastic = TRUE, the number of missing values in miss_cols[i] is a random variable. This random variable is a sum of two binomial distributed variables (one for group 1 and one for group 2). If p is not too high and x is not too high or too low (see below), then the odds 1:x will be met in expectation. But in a single dataset the odds will be unequal to 1:x most of the time.

If p is high and x is too high or too low, it is possible that the odds 1:x and the proportion of missing values p cannot be realized together. For example, if p[i] = 0.9, then a maximum of x = 1.25 is possible (assuming that exactly 50% of the values are below and 50% of the values are above the cutoff value in ctrl_cols[i]). If a combination of p and x that cannot be realized together is given to delete_MAR_1_to_x, then a warning will be generated and x will be adjusted in such a way that p can be realized as given to the function.
delete_MAR_1_to_x

The argument `add_realized_x` controls whether the x of the realized odds are added to the return value or not. If `add_realized_x = TRUE`, then the realized x values for all `miss_cols` will be added as an attribute to the returned object. For `stochastic = TRUE` these realized x will differ from the given x most of the time and will change if the function is rerun without setting a seed. For `stochastic = FALSE`, it is also possible that the realized odds differ (see above). However, the realized odds will be constant over multiple runs.

Value

An object of the same class as `ds` with missing values

Treatment of factors

If `ds[,ctrl_cols[i]]` is an unordered factor, then the concept of a cutoff value is not meaningful and cannot be applied. Instead, a combinations of the levels of the unordered factor is searched that

- guarantees at least a proportion of `prop` rows are in group 1
- minimize the difference between `prop` and the proportion of rows in group 1.

This can be seen as a binary search problem, which is solved by the solver from the package `lpSolve`, if `use_lpSolve = TRUE`. If `use_lpSolve = FALSE`, a very simple heuristic is applied. The heuristic only guarantees that at least a proportion of `prop` rows are in group 1. The choice `use_lpSolve = FALSE` is not recommend and should only be considered, if the solver of lpSolve fails.

If `ordered_as_unordered = TRUE`, then ordered factors will be treated like unordered factors and the same binary search problem will be solved for both types of factors. If `ordered_as_unordered = FALSE` (the default), then ordered factors will be grouped via `cutoff_fun` as described in Details.

References


See Also

- `delete_MNAR_1_to_x`

Other functions to create MAR: `delete_MAR_censoring()`, `delete_MAR_one_group()`, `delete_MAR_rank()`

Examples

ds <- data.frame(X = 1:20, Y = 101:120)
delete_MAR_1_to_x(ds, 0.2, "X", "Y", 3)
# beware of small datasets and stochastic = FALSE
attr(delete_MAR_1_to_x(ds, 0.4, "X", "Y", 3, add_realized_x = TRUE), "realized_x")
attr(delete_MAR_1_to_x(ds, 0.4, "X", "Y", 4, add_realized_x = TRUE), "realized_x")
attr(delete_MAR_1_to_x(ds, 0.4, "X", "Y", 5, add_realized_x = TRUE), "realized_x")
attr(delete_MAR_1_to_x(ds, 0.4, "X", "Y", 7, add_realized_x = TRUE), "realized_x")
# p = 0.4 and 20 values -> 8 missing values, possible combinations:
# either 6 above 2 below (x = 3) or
# 7 above and 1 below (x = 7)
# Too high combination of p and x:
defile_MAR_1_to_x(ds, 0.9, "X", "Y", 3)
defile_MAR_1_to_x(ds, 0.9, "X", "Y", 3, stochastic = TRUE)

---

**delete_MAR_censoring**

*Create MAR values using a censoring mechanism*

**Description**

Create missing at random (MAR) values using a censoring mechanism in a data frame or a matrix.

**Usage**

```r
defile_MAR_censoring(
  ds,
  p,
  miss_cols,
  ctrl_cols,
  where = "lower",
  sorting = TRUE
)
```

**Arguments**

- **ds**: a data frame or matrix in which missing values will be created.
- **p**: a numeric vector with length one or equal to length `miss_cols`; the probability that a value is missing.
- **miss_cols**: a vector of column names or indices of columns in which missing values will be created.
- **ctrl_cols**: a vector of column names or indices of columns, which controls the creation of missing values in `miss_cols`. Must be of the same length as `miss_cols`.
- **where**: controls where missing values are created; one of "lower", "upper" or "both" (see details).
- **sorting**: logical; should sorting be used or a quantile as a threshold.

**Details**

This function creates missing at random (MAR) values in the columns specified by the argument `miss_cols`. The probability for missing values is controlled by `p`. If `p` is a single number, then the overall probability for a value to be missing will be `p` in all columns of `miss_cols`. (Internally `p` will be replicated to a vector of the same length as `miss_cols`. So, all `p[i]` in the following sections will be equal to the given single number `p`.) Otherwise, `p` must be of the same length as `miss_cols`. In this case, the overall probability for a value to be missing will be `p[i]` in the column `miss_cols[i]`. The position of the missing values in `miss_cols[i]` is controlled by `ctrl_cols[i]`. The following procedure is applied for each pair of `ctrl_cols[i]` and `miss_cols[i]` to determine the positions of missing values:
If sorting = TRUE (the default), the column ctrl_cols[i] will be sorted. Then the rows with the round(nrow(ds) * p[i]) smallest values will be selected (if where = "lower" (the default)). Now missing values will be created in the column miss_cols[i] in these rows. This effectively censors the proportion of p[i] rows of smallest values in ctrl_cols[i] in miss_cols[i].

If where = "upper", instead of the rows with the smallest values, the rows with the highest values will be selected. For where = "both", the one half of the round(nrow(ds) * p[i]) rows with missing values will be the rows with the smallest values and the other half will be the rows with the highest values. So the censoring rows are dived to the highest and smallest values of ctrl_cols[i].

If sorting = FALSE, the rows of ds will not be sorted. Instead, a quantile will be calculated (using quantile). If where = "lower", the quantile(ds[,ctrl_cols[i]],p[i]) will be calculated and all rows with values in ds[,ctrl_cols[i]] below this quantile will have missing values in miss_cols[i]. For where = "upper", the quantile(ds[,ctrl_cols[i]],1 - p[i]) will be calculated and all rows with values above this quantile will have missing values. For where = "both", the quantile(ds[,ctrl_cols[i]],p[i] / 2) and quantile(ds[,ctrl_cols[i]],1 - p[i] / 2) will be calculated. All rows with values in ctrl_cols[i] below the first quantile or above the second quantile will have missing values in miss_cols[i].

The option sorting = TRUE will always create exactly round(nrow(ds) * p[i]) missing values in miss_cols[i]. For sorting = FALSE, the number of missing values will normally be close to nrow(ds) * p[i]. But for ctrl_cols with many duplicates the choice sorting = FALSE can be problematic, because of the calculation of quantile(ds[,ctrl_cols[i]],p[i]) and setting values NA below this threshold (see examples). So, in most cases sorting = TRUE is recommended.

Value

An object of the same class as ds with missing values

References


See Also

date_MAR_censoring

Other functions to create MAR: delete_MAR_1_to_x(), delete_MAR_one_group(), delete_MAR_rank()

Examples

d <- data.frame(X = 1:20, Y = 101:120)
date_MAR_censoring(d, 0.2, "X", "Y")
# many dupllicated values can be problematic for sorting = FALSE:
d_many_dup <- data.frame(X = 1:20, Y = c(rep(0, 10), rep(1, 10)))
date_MAR_censoring(d_many_dup, 0.2, "X", "Y") # 4 NAs as expected
quantile(d_many_dup$Y, 0.2) # 0
# No value is BELOW 0 in d_many_dup$Y, so no missing values will be created:
date_MAR_censoring(d_many_dup, 0.2, "X", "Y", sorting = FALSE) # No NA!
Description

Create missing at random (MAR) values by deleting values in one of two groups in a data frame or a matrix

Usage

```r
delete_MAR_one_group(
  ds,
  p,
  miss_cols,
  ctrl_cols,
  cutoff_fun = median,
  prop = 0.5,
  use_lpSolve = TRUE,
  ordered_as_unordered = FALSE,
  stochastic = FALSE,
  ...
)
```

Arguments

- `ds`: a data frame or matrix in which missing values will be created
- `p`: a numeric vector with length one or equal to length `miss_cols`; the probability that a value is missing
- `miss_cols`: a vector of column names or indices of columns in which missing values will be created
- `ctrl_cols`: a vector of column names or indices of columns, which controls the creation of missing values in `miss_cols`. Must be of the same length as `miss_cols`.
- `cutoff_fun`: function that calculates the cutoff values in the `ctrl_cols`
- `prop`: numeric of length one; (minimum) proportion of rows in group 1
- `use_lpSolve`: logical; should lpSolve be used for the determination of groups, if `ctrl_cols[i]` is an unordered factor
- `ordered_as_unordered`: logical; should ordered factors be treated as unordered factors
- `stochastic`: logical; see details
- `...`: further arguments passed to `cutoff_fun`
Details

This function creates missing at random (MAR) values in the columns specified by the argument `miss_cols`. The probability for missing values is controlled by `p`. If `p` is a single number, then the overall probability for a value to be missing will be `p` in all columns of `miss_cols`. (Internally `p` will be replicated to a vector of the same length as `miss_cols`. So, all `p[i]` in the following sections will be equal to the given single number `p`.) Otherwise, `p` must be of the same length as `miss_cols`. In this case, the overall probability for a value to be missing will be `p[i]` in the column `miss_cols[i]`. The position of the missing values in `miss_cols[i]` is controlled by `ctrl_cols[i]`. The following procedure is applied for each pair of `ctrl_cols[i]` and `miss_cols[i]` to determine the positions of missing values:

At first, the rows of `ds` are divided into two groups. Therefore, the `cutoff_fun` calculates a cutoff value for `ctrl_cols[i]` (via `cutoff_fun(ds[,ctrl_cols[i]],...)`). The group 1 consists of the rows, whose values in `ctrl_cols[i]` are below the calculated cutoff value. If the so defined group 1 is empty, the rows that are equal to the cutoff value will be added to this group (otherwise, these rows will belong to group 2). The group 2 consists of the remaining rows, which are not part of group 1. Now one of these two groups is chosen randomly. In the chosen group, values are deleted in `miss_cols[i]`. In the other group, no missing values will be created in `miss_cols[i]`.

If `stochastic = FALSE` (the default), then floor(nrow(ds) * `p[i]`) or ceiling(nrow(ds) * `p[i]`) values will be set NA in column `miss_cols[i]` (depending on the grouping). If `stochastic = TRUE`, each value in the group with missing values will have a probability to be missing, to meet a proportion of `p[i]` of missing values in `miss_cols[i]` in expectation. The effect of `stochastic` is further discussed in `delete_MCAR`.

Value

An object of the same class as `ds` with missing values

Treatment of factors

If `ds[,ctrl_cols[i]]` is an unordered factor, then the concept of a cutoff value is not meaningful and cannot be applied. Instead, a combinations of the levels of the unordered factor is searched that

- guarantees at least a proportion of `prop` rows are in group 1
- minimize the difference between `prop` and the proportion of rows in group 1.

This can be seen as a binary search problem, which is solved by the solver from the package `lpSolve`, if `use_lpSolve = TRUE`. If `use_lpSolve = FALSE`, a very simple heuristic is applied. The heuristic only guarantees that at least a proportion of `prop` rows are in group 1. The choice `use_lpSolve = FALSE` is not recommend and should only be considered, if the solver of `lpSolve` fails.

If `ordered_as_unordered = TRUE`, then ordered factors will be treated like unordered factors and the same binary search problem will be solved for both types of factors. If `ordered_as_unordered = FALSE` (the default), then ordered factors will be grouped via `cutoff_fun` as described in Details.

References

**delete_MAR_rank**

**See Also**

`delete_MNAR_one_group`

Other functions to create MAR: `delete_MAR_1_to_x()`, `delete_MAR_censoring()`, `delete_MAR_rank()`

**Examples**

```r
ds <- data.frame(X = 1:20, Y = 101:120)
delete_MAR_one_group(ds, 0.2, "X", "Y")
```

---

**delete_MAR_rank**  
*Create MAR values using a ranking mechanism*

**Description**

Create missing at random (MAR) values using a ranking mechanism in a data frame or a matrix

**Usage**

```r
delete_MAR_rank(ds, p, miss_cols, ctrl_cols, ties.method = "average")
```

**Arguments**

- `ds` a data frame or matrix in which missing values will be created
- `p` a numeric vector with length one or equal to length `miss_cols`; the probability that a value is missing
- `miss_cols` a vector of column names or indices of columns in which missing values will be created
- `ctrl_cols` a vector of column names or indices of columns, which controls the creation of missing values in `miss_cols`. Must be of the same length as `miss_cols`.
- `ties.method` how ties are handled, passed to `rank`

**Details**

This function creates missing at random (MAR) values in the columns specified by the argument `miss_cols`. The probability for missing values is controlled by `p`. If `p` is a single number, then the overall probability for a value to be missing will be `p` in all columns of `miss_cols`. (Internally `p` will be replicated to a vector of the same length as `miss_cols`. So, all `p[i]` in the following sections will be equal to the given single number `p`.) Otherwise, `p` must be of the same length as `miss_cols`. In this case, the overall probability for a value to be missing will be `p[i]` in the column `miss_cols[i]`. The position of the missing values in `miss_cols[i]` is controlled by `ctrl_cols[i]`. The following procedure is applied for each pair of `ctrl_cols[i]` and `miss_cols[i]` to determine the positions of missing values:

The probability for a missing value in a row of `miss_cols[i]` is proportional to the rank of the value in `ctrl_cols[i]` in the same row. In total `round(nrow(ds) * p[i])` missing values are created in `miss_cols[i]`. The ranks are calculated via `rank`. The argument `ties.method` is directly passed to this function. Possible choices for `ties.method` are documented in `rank`.

---
delete_MCAR

Value

An object of the same class as ds with missing values

References


See Also

rank, delete_MNAR_rank

Other functions to create MAR: delete_MAR_1_to_x(), delete_MAR_censoring(), delete_MAR_one_group()

Examples

ds <- data.frame(X = 1:20, Y = 101:120)
delete_MAR_rank(ds, 0.2, "X", "Y")

dataframe

<table>
<thead>
<tr>
<th>delete_MCAR</th>
<th>Create MCAR values</th>
</tr>
</thead>
</table>

Description

Create missing completely at random (MCAR) values in a data frame or a matrix

Usage

delete_MCAR(
  ds,
  p,
  miss_cols = seq_len(ncol(ds)),
  stochastic = FALSE,
  p_overall = FALSE
)

Arguments

ds a data frame or matrix in which missing values will be created
p a numeric vector with length one or equal to length miss_cols; the probability that a value is missing
miss_cols a vector of column names or indices of columns in which missing values will be created
stochastic logical; see details
p_overall logical; see details
Details

This function creates missing completely at random (MCAR) values in the dataset \( ds \). The proportion of missing values is specified with \( p \). The columns in which missing values are created can be set via \( \text{miss_cols} \). If \( \text{miss_cols} \) is not specified, then missing values are created in every column.

The probability for missing values is controlled by \( p \). If \( p \) is a single number, then the overall probability for a value to be missing will be \( p \) in all columns of \( \text{miss_cols} \). (Internally \( p \) will be replicated to a vector of the same length as \( \text{miss_cols} \). So, all \( p[i] \) in the following sections will be equal to the given single number \( p \).) Otherwise, \( p \) must be of the same length as \( \text{miss_cols} \). In this case, the overall probability for a value to be missing will be \( p[i] \) in the column \( \text{miss_cols}[i] \).

If \( \text{stochastic} = \text{FALSE} \) and \( \text{p-overall} = \text{FALSE} \) (the default), then exactly round(nrow(ds) \* p[i]) values will be set NA in column \( \text{miss_cols}[i] \). If \( \text{stochastic} = \text{FALSE} \) and \( \text{p-overall} = \text{TRUE} \), then \( p \) must be of length one and exactly round(nrow(ds) \* p \* length(\text{miss_cols})) values will be set NA (over all columns in \( \text{miss_cols} \)). This can result in a proportion of missing values in every \( \text{miss_col} \) unequal to \( p \), but the proportion of missing values in all columns together will be close to \( p \).

If \( \text{stochastic} = \text{TRUE} \), then each value in column \( \text{miss_cols}[i] \) has the probability \( p[i] \) to be missing. In this case, the number of missing values in \( \text{miss_cols}[i] \) is a random variable with a binomial distribution \( B(nrow(ds), p[i]) \). This can (and will most of the time) lead to more or less missing values than round(nrow(ds) \* p[i]) in each column. If \( \text{stochastic} = \text{TRUE} \), then the argument \( \text{p-overall} \) is ignored because it is superfluous.

Value

An object of the same class as \( ds \) with missing values

References


Examples

ds <- data.frame(X = 1:20, Y = 101:120)
delete_MCAR(ds, 0.2)

\[
delete_{-}MNAR\_1\_to\_x
\]

Create MNAR values using MNAR1:x

Description

Create missing not at random (MNAR) values using MNAR1:x in a data frame or a matrix
Usage

delete_MNAR_1_to_x(
  ds,
  p,
  miss_cols,
  x,
  cutoff_fun = median,
  prop = 0.5,
  use_lpSolve = TRUE,
  ordered_as_unordered = FALSE,
  stochastic = FALSE,
  add_realized_x = FALSE,
  ...
)

Arguments

ds         a data frame or matrix in which missing values will be created
p           a numeric vector with length one or equal to length miss_cols; the probability that a value is missing
miss_cols   a vector of column names or indices of columns in which missing values will be created
x           numeric with length one (0 < x < Inf); odds are 1 to x for the probability of a value to be missing in group 1 against the probability of a value to be missing in group 2 (see details)
cutoff_fun  function that calculates the cutoff values in the ctrl_cols
prop        numeric of length one; (minimum) proportion of rows in group 1
use_lpSolve logical; should lpSolve be used for the determination of groups, if ctrl_cols[i] is an unordered factor
ordered_as_unordered logical; should ordered factors be treated as unordered factors
stochastic  logical; see details
add_realized_x logical; if TRUE the realized odds for miss_cols will be returned (as attribute)
...         further arguments passed to cutoff_fun

Details

The functions delete_MNAR_1_to_x and delete_MAR_1_to_x are sisters. The only difference between these two functions is the column that controls the generation of missing values. In delete_MAR_1_to_x a separate column ctrl_cols[i] controls the generation of missing values in miss_cols[i]. In contrast, in delete_MNAR_1_to_x the generation of missing values in miss_cols[i] is controlled by miss_cols[i] itself. All other aspects are identical for both functions. Therefore, further details can be found in delete_MAR_1_to_x.

Value

An object of the same class as ds with missing values
References


See Also

delete_MAR_1_to_x

Other functions to create MNAR: delete_MNAR_censoring(), delete_MNAR_one_group(), delete_MNAR_rank()

Examples

ds <- data.frame(X = 1:20, Y = 101:120)
dele te_MAR_1_to_x(ds, 0.2, "X", x = 3)

---

delete_MNAR_censoring Create MNAR values using a censoring mechanism

Description

Create missing not at random (MNAR) values using a censoring mechanism in a data frame or a matrix

Usage

delete_MNAR_censoring(ds, p, miss_cols, where = "lower", sorting = TRUE)

Arguments

ds a data frame or matrix in which missing values will be created
p a numeric vector with length one or equal to length miss_cols; the probability that a value is missing
miss_cols a vector of column names or indices of columns in which missing values will be created
where controls where missing values are created; one of "lower", "upper" or "both" (see details)
sorting logical; should sorting be used or a quantile as a threshold

Details

The functions delete_MNAR_censoring and delete_MAR_censoring are sisters. The only difference between these two functions is the column that controls the generation of missing values. In delete_MAR_censoring a separate column ctrl_cols[i] controls the generation of missing values in miss_cols[i]. In contrast, in delete_MNAR_censoring the generation of missing values in miss_cols[i] is controlled by miss_cols[i] itself. All other aspects are identical for both functions. Therefore, further details can be found in delete_MAR_censoring.
**Value**

An object of the same class as `ds` with missing values

**References**


**See Also**

`delete_MAR_censoring`

Other functions to create MNAR: `delete_MNAR_1_to_x()`, `delete_MNAR_one_group()`, `delete_MNAR_rank()`

**Examples**

```r
ds <- data.frame(X = 1:20, Y = 101:120)
delete_MAR_censoring(ds, 0.2, "X")
```

---

**delete_MNAR_one_group**  
*Create MNAR values by deleting values in one of two groups*

**Description**

Create missing not at random (MNAR) values by deleting values in one of two groups in a data frame or a matrix

**Usage**

```r
define_MNAR_one_group(
  ds,
  p,
  miss_cols,
  cutoff_fun = median,
  prop = 0.5,
  use_lpSolve = TRUE,
  ordered_as_unordered = FALSE,
  stochastic = FALSE,
  ...
)
```

**Arguments**

- `ds`  
a data frame or matrix in which missing values will be created
- `p`  
a numeric vector with length one or equal to length `miss_cols`; the probability that a value is missing
- `miss_cols`  
a vector of column names or indices of columns in which missing values will be created
The functions `delete_MNAR_one_group` and `delete_MAR_one_group` are sisters. The only difference between these two functions is the column that controls the generation of missing values. In `delete_MAR_one_group` a separate column `ctrl_cols[i]` controls the generation of missing values in `miss_cols[i]`. In contrast, in `delete_MNAR_one_group` the generation of missing values in `miss_cols[i]` is controlled by `miss_cols[i]` itself. All other aspects are identical for both functions. Therefore, further details can be found in `delete_MAR_one_group`.

### Value

An object of the same class as `ds` with missing values

### References


### See Also

- `delete_MAR_one_group`
- Other functions to create MNAR: `delete_MNAR_1_to_x()`, `delete_MNAR_censoring()`, `delete_MNAR_rank()`

### Examples

```r
ds <- data.frame(X = 1:20, Y = 101:120)
delete_MNAR_one_group(ds, 0.2, "X")
```

---

**delete_MNAR_rank**  
Create MNAR values using a ranking mechanism

**Description**

Create missing not at random (MNAR) values using a ranking mechanism in a data frame or a matrix.
Usage

```r
delete_MNAR_rank(ds, p, miss_cols, ties.method = "average")
```

Arguments

- `ds`: a data frame or matrix in which missing values will be created
- `p`: a numeric vector with length one or equal to length `miss_cols`; the probability that a value is missing
- `miss_cols`: a vector of column names or indices of columns in which missing values will be created
- `ties.method`: how ties are handled, passed to `rank`

Details

The functions `delete_MNAR_rank` and `delete_MAR_rank` are sisters. The only difference between these two functions is the column that controls the generation of missing values. In `delete_MAR_rank` a separate column `ctrl_cols[i]` controls the generation of missing values in `miss_cols[i]`. In contrast, in `delete_MNAR_rank` the generation of missing values in `miss_cols[i]` is controlled by `miss_cols[i]` itself. All other aspects are identical for both functions. Therefore, further details can be found in `delete_MAR_rank`.

Value

An object of the same class as `ds` with missing values

References


See Also

- `delete_MAR_rank`
- Other functions to create MNAR: `delete_MNAR_1_to_x()`, `delete_MNAR_censoring()`, `delete_MNAR_one_group()`

Examples

```r
ds <- data.frame(X = 1:20, Y = 101:120)
delete_MNAR_rank(ds, 0.2, "X")
```
### evaluate_imputation_parameters

*Evaluate estimated parameters after imputation*

#### Description

Compares estimated parameters after imputation to true parameters or estimates based on the original dataset.

#### Usage

```r
evaluate_imputation_parameters(
  imp_ds,
  orig_ds = NULL,
  true_pars = NULL,
  parameter = "mean",
  criterion = "RMSE",
  which_cols = seq_len(ncol(imp_ds)),
  tolerance = sqrt(.Machine$double.eps),
  ...
)
```

#### Arguments

- `imp_ds`: a data frame or matrix with imputed values.
- `orig_ds`: a data frame or matrix with original (true) values.
- `true_pars`: true parameters, normally a vector or a matrix.
- `parameter`: a string specifying the estimated parameters for comparison.
- `criterion`: a string specifying the used criterion for comparing the imputed and original values.
- `which_cols`: indices or names of columns used for evaluation.
- `tolerance`: numeric, only used for `criterion = "precision"`: numeric differences smaller than tolerance are treated as zero/equal.
- `...`: further arguments passed to function for parameter estimation.

#### Details

Either `orig_ds` or `true_pars` must be supplied and the other one must be `NULL` (default: both are `NULL`, just supply one, see examples). The following parameters are implemented: "mean", "median", "var", "sd", "quantile", "cov", "cor". Some details follows:

- "var": only the variances of the columns (the diagonal elements of the covariance matrix) are compared. The whole covariance matrix can be compared with "cov".
- "quantile": the quantiles can be set via the additional argument `probs` (see examples). Otherwise, the default quantiles from `quantile` will be used.
The argument `which_cols` allows the selection of columns for comparison (see examples). If `true_pars` is used, it is assumed that only relevant parameters are supplied (see examples). Possible choices for the argument `criterion` are documented in `evaluate_imputed_values`.

**Value**

a numeric vector of length one

**References**


**See Also**

Other evaluation functions: `evaluate_imputed_values()`, `evaluate_parameters()`

**Examples**

```
# only orig_ds known
orig_ds <- data.frame(X = 1:10, Y = 101:101)
imp_ds <- impute_mean(delete_MCAR(orig_ds, 0.4))
evaluate_imputation_parameters(imp_ds, orig_ds = orig_ds)

# true parameters known
orig_ds <- data.frame(X = rnorm(100), Y = rnorm(100, mean = 10))
imp_ds <- impute_mean(delete_MCAR(orig_ds, 0.3))
evaluate_imputation_parameters(imp_ds, true_pars = c(0, 10), parameter = "mean")
evaluate_imputation_parameters(imp_ds, true_pars = c(1, 1), parameter = "var")

# set quantiles
evaluate_imputation_parameters(imp_ds,
   true_pars = c(qnorm(0.3), qnorm(0.3, mean = 10)),
   parameter = "quantile", probs = 0.3)

# compare only column Y
evaluate_imputation_parameters(imp_ds,
   true_pars = c(Y = 10), parameter = "mean",
   which_cols = "Y")
```

---

**evaluate_imputed_values**

*Evaluate imputed values*

**Description**

Compare imputed to true values
evaluate_imputed_values

Usage

evaluate_imputed_values(
    imp_ds,
    orig_ds,
    criterion = "RMSE",
    M = NULL,
    which_cols = seq_len(ncol(imp_ds)),
    tolerance = sqrt(.Machine$double.eps)
)

Arguments

imp_ds    a data frame or matrix with imputed values
orig_ds   a data frame or matrix with original (true) values
criterion a string specifying the used criterion for comparing the imputed and original
           values
M         NULL (the default) or a missing data indicator matrix; the missing data indicator
           matrix is normally created via is.na(miss_ds), where miss_ds is the dataset
           after deleting values from orig_ds
which_cols indices or names of columns used for evaluation
tolerance numeric, only used for criterion = "precision": numeric differences smaller
           than tolerance are treated as zero/equal

Details

The following criterions are implemented to compare the imputed values to the true values:

- "RMSE" (the default): The Root Mean Squared Error between the imputed and true values
- "bias": The mean difference between the imputed and the true values
- "cor": The correlation between the imputed and true values
- "MAE": The Mean Absolute Error between the imputed and true values
- "MSE": The Mean Squared Error between the imputed and true values
- "NRMSE_col_mean": For every column the RMSE divided by the mean of the true values is
  calculated. Then these columnwise values are squared and averaged. Finally, the square root
  of this average is returned.
- "NRMSE_col_mean_sq": For every column the RMSE divided by the square root of the mean
  of the squared true values is calculated. Then these columnwise values are squared and averaged.
  Finally, the square root of this average is returned.
- "NRMSE_col_sd": For every column the RMSE divided by the standard deviation of all true
  values is calculated. Then these columnwise values are squared and averaged. Finally, the
  square root of this average is returned.
- "NRMSE_tot_mean": RMSE divided by the mean of all true values
- "NRMSE_tot_mean_sq": RMSE divided by the square root of the mean of all squared true
  values
evaluate_imputed_values

- "NRMSE_tot_sd": RMSE divided by the standard deviation of all true values
- "nr_equal": number of imputed values that are equal to the true values
- "nr_NA": number of values in imp_ds that are NA (not imputed)
- "precision": proportion of imputed values that are equal to the true values

Additionally there are relative versions of bias and MAE implemented. In the relative versions, the differences are divided by the absolute values of the true values. These relative versions can be selected via "bias_rel" and "MAE_rel". The "NRMSE_tot_*" and "NRMSE_col_*" are equal, if the columnwise normalization values are equal to the total normalization value (see examples).

The argument which_cols allows the selection of columns for comparison (see examples).

If M = NULL (the default), then all values of imp_ds and orig_ds will be used for the calculation of the evaluation criterion. If a missing data indicator matrix is given via M, only the truly imputed values (values that are marked as missing via M) will be used for the calculation. If you want to provide M, M must be a logical matrix of the same dimensions as orig_ds and missing values must be coded as TRUE. This is the standard behavior, if you use is.na on a dataset with missing values to generate M (see examples). It is possible to combine M and which_cols.

Value

a numeric vector of length one

References


See Also

Other evaluation functions: evaluate_imputation_parameters(), evaluate_parameters()

Examples

```r
orig_ds <- data.frame(X = 1:10, Y = 101:110)
miss_ds <- delete_MCAR(orig_ds, 0.3)
imp_ds <- impute_mean(miss_ds)
# compare all values from orig_ds and imp_ds
evaluate_imputed_values(imp_ds, orig_ds)
# compare only the imputed values
M <- is.na(miss_ds)
evaluate_imputed_values(imp_ds, orig_ds, M = M)
# compare only the imputed values in column X
evaluate_imputed_values(imp_ds, orig_ds, M = M, which_cols = "X")

# NRMSE_tot_mean and NRMSE_col_mean are equal, if columnwise means are equal
orig_ds <- data.frame(X = 1:10, Y = 10:1)
miss_ds <- delete_MCAR(orig_ds, 0.3)
imp_ds <- impute_mean(miss_ds)
evaluate_imputed_values(imp_ds, orig_ds, "NRMSE_tot_mean")
evaluate_imputed_values(imp_ds, orig_ds, "NRMSE_col_mean")
```
**evaluate_parameters**

Evaluate estimated parameters

**Description**

Compare estimated parameters to true parameters

**Usage**

```r
evaluate_parameters(
  est_pars,
  true_pars,
  criterion = "RMSE",
  tolerance = sqrt(.Machine$double.eps)
)
```

**Arguments**

- `est_pars`: a vector or matrix of estimated parameters
- `true_pars`: true parameters, normally a vector or a matrix
- `criterion`: a string specifying the used criterion for comparing the imputed and original values
- `tolerance`: numeric, only used for `criterion = "precision"`: numeric differences smaller than tolerance are treated as zero/equal

**Details**

The same criteria are implemented for `evaluate_parameters` and `evaluate_imputed_values`. The possible choices are documented in `evaluate_imputed_values`.

**Value**

a numeric vector of length one

**References**


**See Also**

Other evaluation functions: `evaluate_imputation_parameters()`, `evaluate_imputed_values()`

**Examples**

```r
evaluate_parameters(1:4, 2:5, "RMSE")
```
impute_mean  

Mean imputation

Description

Impute an observed mean for the missing values

Usage

impute_mean(ds, type = "columnwise")

Arguments

- **ds**: a data frame or matrix with missing values
- **type**: one of: "columnwise", "rowwise", "total", "Two-Way" or "Winer" (see details)

Details

For every missing value the mean of some observed values is imputed. The observed values to be used are specified via `type`. For example, `type = "columnwise"` (the default) imputes the mean of the observed values in a column for all missing values in the column. This is normally meant, if someone speaks of "imputing the mean" or "mean imputation".

Other options for `type` are: "rowwise", "total", "Winer" and "Two-way". The option "rowwise" imputes all missing values in a row with the mean of the observed values in the same row. "total" will impute every missing value with the mean of all observed values in `ds`. "Winer" imputes the mean of the rowwise and columnwise mean. Beland et al. (2016) called this method "Winer" and they attributed the method to Winer (1971). "Two-way" imputes the sum of rowwise and columnwise mean minus the total mean. This method was suggested by D.B Rubin to Bernaards & Sijtsma, K. (2000).

Value

An object of the same class as `ds` with imputed missing values

A Note for tibble users

If you use tibbles and an error like ‘Lossy cast from ‘value’<double> to ‘x’<integer>’ occurs, you will first need to convert all integer columns with missing values to double. Another solution is to convert the tibble with `as.data.frame()` to a data frame. The data frame will automatically convert integer columns to double columns, if needed.

References

impute_median


See Also

apply_imputation the workhorse for this function

Other location parameter imputation functions: impute_median(), impute_mode()

Examples

ds <- data.frame(X = 1:20, Y = 101:120)
ds_miss <- delete_MCAR(ds, 0.2)
ds_imp <- impute_mean(ds_miss)
# completely observed columns can be of any type:
ds_miss_char <- cbind(ds_miss, letters[1:20])
ds_imp_char <- impute_mean(ds_miss_char)

---

impute_median

Median imputation

Description

Impute an observed median value for every missing value

Usage

impute_median(ds, type = "columnwise", ordered_low = FALSE)

Arguments

ds a data frame or matrix with missing values
type one of: "columnwise", "rowwise", "total", "Two-Way" or "Winer" (see details)
ordered_low logical; used for the calculation of ordered factors (for details see: median.factor)

Details

This function behaves exactly like impute_mean. The only difference is that it imputes a median instead of a mean. All types from impute_mean are also implemented for impute_median. They are documented in impute_mean and apply_imputation. The function median is used for the calculation of the median values for imputation.

Value

An object of the same class as ds with imputed missing values
A Note for tibble users

If you use tibbles and an error like 'Lossy cast from 'value' <double> to 'x' <integer>' occurs, you will first need to convert all integer columns with missing values to double. Another solution is to convert the tibble with as.data.frame() to a data frame. The data frame will automatically convert integer columns to double columns, if needed.

References


See Also

apply_imputation (the workhorse for this function)
median, median.factor

Other location parameter imputation functions: impute_mean(), impute_mode()

Examples

ds <- data.frame(X = 1:20, Y = ordered(LETTERS[1:20]))
ds_miss <- delete_MCAR(ds, 0.2)
ds_imp <- impute_median(ds_miss)
  # completely observed columns can be of any type:
ds_miss_char <- cbind(ds_miss, letters[1:20])
ds_imp_char <- impute_median(ds_miss_char)

impute_mode

Mode imputation

Description

Impute an observed mode value for every missing value

Usage

impute_mode(ds, type = "columnwise")

Arguments

ds a data frame or matrix with missing values
type one of: "columnwise", "rowwise", "total", "Two-Way" or "Winer" (see details)

Details

This function behaves exactly like impute_mean. The only difference is that it imputes a mode instead of a mean. All types from impute_mean are also implemented for impute_mode. They are documented in impute_mean and apply_imputation.

A mode value of a vector \( x \) is a most frequent value of \( x \). If this value is not unique, the first occurring mode value in \( x \) will be used as imputation value.
**Value**

An object of the same class as `ds` with imputed missing values

**References**


**See Also**

`apply_imputation` the workhorse for this function

Other location parameter imputation functions: `impute_mean()`, `impute_median()`

**Examples**

```r
data.frame(X = c(1:12, rep(8, 8)), Y = 101:120)
data.frame(X = c(1:12, rep(8, 8)), Y = 101:120)
data.frame(X = c(1:12, rep(8, 8)), Y = 101:120)
```

---

**impute_sRHD**

*Simple random hot deck imputation*

**Description**

Impute missing values in a data frame or a matrix using a simple random hot deck

**Usage**

```r
impute_sRHD(ds, type = "cols_seq", donor_limit = Inf)
```

**Arguments**

- **ds** a data frame or matrix with missing values
- **type** the type of hot deck; the default ("cols_seq") is a random hot deck that imputes each column separately. Other choices are "sim_comp" and "sim_part". Both impute all missing values in an object (row) simultaneously using a single donor object. The difference between the two types is the choice of objects that can act as donors. "sim_comp:" only completely observed objects can be donors. "sim_part": all objects that have no missing values in the missing parts of a recipient can be donors.
- **donor_limit** numeric of length one or "min": how many times an object can be a donor. default is Inf (no restriction).
Details

There are three types of simple random hot decks implemented. They can be selected via type:

- "cols_seq" (the default): Each variable (column) is handled separately. If an object (row) has a missing value in a variable (column), then one of the observed values in the same variable is chosen randomly and the missing value is replaced with this chosen value. This is done for all missing values.

- "sim_comp": All missing variables (columns) of an object are imputed together ("simultaneous"). For every object with missing values (such an object is called a recipient in hot deck terms), one complete object is chosen randomly and all missing values of the recipient are imputed with the values from the complete object. A complete object used for imputation is called a donor.

- "sim_part": All missing variables (columns) of an object are imputed together ("simultaneous"). For every object with missing values (recipient) one donor is chosen. The donor must have observed values in all the variables that are missing in the recipient. The donor is allowed to have unobserved values in the non-missing parts of the recipient. So, in contrast to "sim_comp", the donor can be partly incomplete.

The parameter donor_limit controls how often an object can be a donor. This parameter is only implemented for types "cols_seq" and "sim_comp". If type = "sim_part" and donor_limit is not Inf, then an error will be thrown. For "sim_comp" the default value (Inf) allows every object to be a donor for an infinite number of times (there is no restriction on the times an object can be a donor). If a numeric value less than Inf is chosen, then every object can be a donor at most donor_limit times. For example donor_limit = 1 ensures that every object donates at most one time. If there are only few complete objects and donor_limit is set too low, then an imputation might not be possible with the chosen donor_limit. In this case, the donor_limit will be adjusted (see examples). Setting donor_limit = "min" chooses automatically the minimum value for donor_limit that allows imputation of all missing values. For type = "cols_seq" the donor limit is applied for every column separately.

Value

An object of the same class as ds with imputed missing values

References


Examples

ds <- data.frame(X = 1:20, Y = 101:120)
ds_miss <- delete_MCAR(ds, 0.2)
ds_imp <- impute_sRHD(ds_miss)

# Warning: donor limit to low
ds_miss_one_donor <- ds
  ds_miss_one_donor[1:19, "X"] <- NA
  impute_sRHD(ds_miss_one_donor, donor_limit = 3)
**median.factor**  

*Median for ordered factors*

**Description**

Compute the median of an ordered factor

**Usage**

```r
## S3 method for class 'factor'
median(x, na.rm = FALSE, ordered_low = FALSE, ...)
```

**Arguments**

- `x`: an ordered factor (for unordered factors an error will be thrown)
- `na.rm`: logical; should `NA` be removed before computation
- `ordered_low`: logical; only used if the length of `x` is even and the two middle values are unequal (see details)
- `...`: not used in this function

**Details**

Currently, the median for an ordered factor is not implemented in base R. This function is a remedy for this. It allows the computation of “a median” for ordered factors (see below) and overwrites the error message for unordered factors from `median.default` (hence, the function name is `median.factor` and not `median.ordered`).

If the length of `x` is even, then the median will be the middle value of the sorted list of elements from `x`. If the length of `x` is odd and the two middle values of the sorted list of elements from `x` are equal, then the median is one of these (equal) middle values. The only problematic case is an odd length `x` with unequal middle values. In this case, the median of a numeric vector is normally defined as the mean of the two middle values. However, for ordered factors the mean is not defined. The argument `ordered_low` cures this problem. If `ordered_low = FALSE` (the default), then the larger of the two middle values is returned (this value is called ‘hi-median’ in `mad`). If `ordered_low = TRUE`, then the smaller of the two middle values is returned (this value is called ‘low-median’ in `mad`).

**Value**

a length-one factor

**Examples**

```r
ord_factor_odd <- ordered(letters[1:5])
median(ord_factor_odd) # calls median.factor, if package is loaded

# if only base R is loaded, median.default will be called and will throw an error:
tryCatch(median.default(ord_factor_odd), error = function(e) e)
```
ord_factor_even <- ordered(letters[1:4])
median(ord_factor_even, ordered_low = FALSE)
median(ord_factor_even, ordered_low = TRUE)
Index

apply, 3
apply_imputation, 2, 25–27

delete_MAR_1_to_x, 4, 8, 11, 12, 14, 15
delete_MAR_censoring, 6, 7, 11, 12, 15, 16
delete_MAR_one_group, 6, 8, 9, 12, 17
delete_MAR_rank, 6, 8, 11, 11, 18
delete_MCAR, 10, 12
delete_MNAR_1_to_x, 6, 13, 16–18
delete_MNAR_censoring, 8, 15, 15, 17, 18
delete_MNAR_one_group, 11, 15, 16, 16, 18
delete_MNAR_rank, 12, 15–17, 17

evaluate_imputation_parameters, 19, 22, 23
evaluate_imputed_values, 20, 20, 23
evaluate_parameters, 20, 22, 23

impute_mean, 3, 24, 25–27
impute_median, 3, 25, 25, 27
impute_mode, 3, 25, 26, 26
impute_sRHD, 27
is.na, 22

mad, 29
median, 25, 26
median.default, 29
median.factor, 25, 26, 29

quantile, 8, 19

rank, 11, 12, 18