Package ‘mi’

July 22, 2025

Type Package

Title Missing Data Imputation and Model Checking

Version 1.1

Date 2022-06-05

Description
The mi package provides functions for data manipulation, imputing missing values in an approximate Bayesian framework, diagnostics of the models used to generate the imputations, confidence-building mechanisms to validate some of the assumptions of the imputation algorithm, and functions to analyze multiply imputed data sets with the appropriate degree of sampling uncertainty.

VignetteBuilder knitr

Depends R (>= 3.0.0), methods, Matrix, stats4

Imports arm (>= 1.4-11)

Suggests betareg, lattice, knitr, MASS, nnet, parallel, sn, survival, truncnorm, foreign

URL http://www.stat.columbia.edu/~gelman/

License GPL (>= 2)

LazyLoad yes

Author Andrew Gelman [ctb],
Jennifer Hill [ctb],
Yu-Sung Su [aut],
Masanao Yajima [ctb],
Maria Pittau [ctb],
Ben Goodrich [cre, aut],
Yajuan Si [ctb],
Jon Kropko [aut]

Maintainer Ben Goodrich <benjamin.goodrich@columbia.edu>

NeedsCompilation no

Repository CRAN

Date/Publication 2022-06-06 20:10:04 UTC

Contents

`00mi-package` *Iterative Multiple Imputation from Conditional Distributions*

Description

The mi package performs multiple imputation for data with missing values. The algorithm iteratively draws imputed values from the conditional distribution for each variable given the observed and imputed values of the other variables in the data. The process approximates a Bayesian framework; multiple chains are run and convergence is assessed after a pre-specified number of iterations within each chain. The package allows customization of the conditional model and the treatment of missing values for each variable. In addition, the package provides graphics to visualize missing data patterns, to diagnose the models used to generate the imputations, and to assess convergence. Functions are included to run statistical models post-imputation with the appropriate degree of sampling uncertainty.

Details

Package:	mi
Type:	Package
Version:	1.0
Date:	Sun Jun 5 01:30:56 2022
License:	GPL (>= 2)
LazyLoad:	yes

See the vignette for an example of typical usage.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima,Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_data.frame`, `change`, `mi`, `Rhats`, `pool`, `complete`

01missing_variable *Class "missing_variable" and Inherited Classes*

Description

The missing_variable class is essentially the data comprising a variable plus all the metadata needed to understand how its missing values will be imputed. However, no variable is merely of missing_variable class; rather every variable is of a class that inherits from the missing_variable class. Even if a variable has no missing values, it needs to be coerced to a class that inherits from the missing_variable class before it can be used to impute values of other missing_variables. Understanding the properties of different subclasses of the missing_variable class is essential for modeling and imputing them. The `missing_data.frame-class` is essentially a list of objects that inherit from the missing_variable class, plus metadata need to understand how these missing_variables relate to each other. Most users will never need to call `missing_variable` directly since it is called by `missing_data.frame`.

Usage

```
missing_variable(y, type, ...)
## Hidden arugments not included in the signature:
## favor_ordered = TRUE, favor_positive = FALSE,
## variable_name = deparse(substitute(y))
```

Arguments

`y`	Can be any vector, some of whose values may be `NA`, which will comprise the **raw_data** slot of a missing_variable (see the Slots section). It is recommended that this vector *not* have any transformations, such as a log-transformation. Any continuous variable can be transformed using the function in its **transformation** slot. The transformations and other discretionary aspects of a missing_variable are typically changed by calling the `change` function on a `missing_data.frame` See the Slots section for more details.
`type`	Missing or a character string among the classes that inherit from the missing_variable class. If missing, the constructor will guess (sometimes incorrectly) based on the characteristics of the variable. The best way to improve the guessing of categorical variables is to use the `factor` function — possibly with `ordered = TRUE` — to create (possibly ordered) factors that will correctly be coerced to objects of `unordered-categorical-class` and `ordered-categorical-class` respectively. If you fail to do so, the hidden arguments that are not included in the signature affect the guesses. If `favor_ordered = TRUE`, which is the default, it will tend to guess that variables with few unique values are should be coerced to `ordered-categorical-class` and `unordered-categorical-class` otherwise. If `favor_positive = FALSE`, which is the default, it will tend to guess that variables with many unique values are merely continuous, whether or not all the observed values are positive. If `favor_positive = TRUE` nonnegative or positive variables will get coerced to `nonnegative-continuous-class` or `positive-continuous-class`. See the Slots section and the specific help pages for more details on the subclasses.
`...`	Further hidden arguments that are not in the signature. The `favor_ordered` and `favor_positive` arguments are documented immediately above. The `variable` name argument can be used to control what gets put in the **variable_name** slot, see the Slots section below.

Value

The missing_variable function returns an object that inherits from the missing_variable class.

Objects from the Classes

The missing_variable class is virtual, so no objects may be created from it. However, the missing_variable generic function can be used to instantiate an object that inherits from the missing_variable class by specifying its `type` argument. A user would call the `missing_data.frame` function on a `data.frame`, which in turn calls the missing_variable function on each column of the `data.frame` using various heuristics to guess the `type` argument.

Slots

In the following table, indentation indicates inheritance from the class with less indentation, and italics indicates that the class is virtual so no variables can be created with that class. Inherited classes inherit the transformations, families, link functions, and `fit_model-methods` from their parent class, although these are often superceeded by analogues that are tailored for the inherited class. Also note, the default transformation for the continuous class is a standardization using *twice* the standard deviation of the observed values.

The distinction between the transformation entailed by the `family` and the transformation entailed by the function in the **tranformation** slot may be confusing at this point. The former pertains to how the linear predictor of a variable is mapped to the space of a variable when it is on the left-hand side of a generalized linear model. The latter pertains — for continuous variables only — to how the values in the **raw_data** slot are mapped into those in the **data** and thus affects how a continuous variable enters into the model whether it is on the left or right-hand side. The classes are discussed in much more detail below.

Class name [transformation]	**Default family and link**	**Default `fit_model`**
missing_variable	none	throws error
categorical	none	throws error
unordered-categorical	`binomial(link = 'logit')`	`multinom`
ordered-categorical	`binomial(link = 'logit')`	`bayespolr`
binary	`binomial(link = 'logit')`	`bayesglm`
interval	`gaussian{link = 'identity'}`	`survreg`
continuous[standardize]	`gaussian{link = 'identity'}`	`bayesglm`
semi-continuous[identity]		
nonnegative-continuous[logshift]		
SC_proportion[squeeze]	`binomial(link = 'logit')`	`betareg`
positive-continuous[`log`]		
proportion[identity]	`binomial(link = 'logit')`	`betareg`
bounded-continuous[identity]		
count	`quasipoisson{link = 'log'}`	`bayesglm`
irrelevant		throws error
fixed		throws error

The missing_variable class is virtual and has the following slots (this information is primarily directed at developeRs):

`variable_name`**:** Object of class `character` of length one naming the variable

`raw_data`**:** Object of class `"ANY"` representing the observations on a variable, some of which may be `NA`. No method should ever change this slot at all. Instead, methods should change the **data** slot.

`data`**:** Object of class `"ANY"`, which is initially a copy of the **raw_data** slot — transformed by the function in the **transformation** slot for continuous variables only — and whose `NA` values are replaced during the multiple imputation process. See `mi`

`n_total`**:** Object of class `"integer"` which is the `length` of the **data** slot

`all_obs`**:** Object of class `"logical"` of length one indicating whether all values of the **data** slot are observed and thus not `NA`

`n_obs`**:** Object of class `"integer"` of length one indicating the number of values of the **data** slot that are observed and thus not `NA`

`which_obs`**:** Object of class `"integer"`, which is a vector indicating the positions of the observed values in the **data** slot `all_miss`**:** Object of class `"logical"` of length one indicating whether all values of the **data** slot are `NA`

`n_miss`**:** Object of class `"integer"` of length one indicating the number of values of the **data** slot that are `NA`

`which_miss`**:** Object of class `"integer"`, which is a vector indicating the positions of the missing values in the **data** slot

`n_extra`**:** Object of class `"integer"` of length one indicating how many (missing) observations have been added to the end of the **data** slot that are not included in the **raw_data** slot. Although the extra values will be imputed, they are not considered to be "missing" for the purposes of defining the previous three slots

`which_extra`**:** Object of class `"integer"`, which is a vector indicating the positions of the extra values at the end of the **data** slot

`n_unpossible`**:** Object of class `"integer"` of length one indicating the number of values that are logically or structurally unobservable

`which_unpossible`**:** Object of class `"integer"` indicating the positions of the unpossible values in the **data** slot

`n_drawn`**:** Object of class `"integer"` of length one which is the sum of the **n_miss** and **n_extra** slots

`which_drawn`**:** Object of class `"integer"` which is a vector concatinating the **which_miss** and **which_extra** slots

`imputation_method`**:** Object of class `"character"` of length one indicating how the `NA` values are to be imputed. Possibilities include "ppd" for imputation from the posterior predictive distribution, "pmm" for imputation via predictive mean matching, "mean" for mean-imputation, "median" for median-imputation, "expectation" for conditional mean-imputation. With enough programming effort, other kinds of imputation can be defined and specified here.

`family`**:** Object of class `"WeAreFamily"` that will typically be passed to `glm` and similar functions during the multiple imputation process

`known_families`**:** Object of class `character` indicating the families that are known to be supported for a class; see `family`

`known_links`**:** Object of class `character` indicating what link functions are known to be supported by the elements of the **known_families** slot; see `family`

`imputations`**:** Object of class `"MatrixTypeThing"` with rows equal to the number of iterations (initially zero) of the multiple imputation algorithm and columns equal to the **n_drawn** slot. The rows are appropriately extended and then filled by the `mi` function

`done`**:** Object of class `"logical"` of length one indicating whether the `NA` values in the **data** slot have been replaced by imputed values

`parameters`**:** Object of class `"MatrixTypeThing"` with rows equal to the number of iterations (initially zero) of the multiple imputation algorithm and columns equal to the number of estimated parameters when modeling the **data** slot. The rows are appropriately extended and then filled by the `mi` function

`model`**:** Object of class `"ANY"` which can be filled by an object that is output by one of the `fit_model-methods`, which is done by default by `mi` when all the iterations have completed `fitted`**:** Object of class `"ANY"` although typically a vector or matrix that contains the fitted values of the model in the slot immediately above. Note that the **fitted** slot is filled by default by `mi`, although the **model** slot is left empty by default to save RAM.

`estimator`**:** Object of class `"character"` of length one indicating which pre-existing `fit_model` to use for an unordered-categorical variable. Options are `"mnl"`, in which `multinom` from the **nnet** package is used to fit the values of the unordered categorical variable; and `"rnl"`, in which each category is separately modeled as the positive binary outcome against all other categories using a `bayesglm` `fit_model` and the probabilities of each category are normalized to sum to 1 after each model is run. In general, `"rnl"` is slightly less accurate than `"mnl"`, but runs much more quickly especially when the unordered categorical variable has many unique categories.

The WeAreFamily class is a class union of `character` and `family`, while the MatrixTypeThing class is a class union of `matrix` only at the moment.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_data.frame`, `categorical-class`, `unordered-categorical-class`, `ordered-categorical-class`, `binary-class`, `interval-class`, `continuous-class`, `semi-continuous-class`, `nonnegative-continuous-class`, `SC_proportion-class`, `censored-continuous-class`, `truncated-continuous-class`, `bounded-continuous-class`, `positive-continuous-class`, `proportion-class`, `count-class`

Examples

```
# STEP 0: GET DATA
data(nlsyV, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
income <- missing_variable(nlsyV$income, type = "continuous")
show(income)

# STEP 1: CONVERT IT TO A missing_data.frame
mdf <- missing_data.frame(nlsyV) # this calls missing_variable() internally
show(mdf)
```

`02missing_data.frame` *Class "missing_data.frame"*

Description

This class is similar to a `data.frame` but is customized for the situation in which variables with missing data are being modeled for multiple imputation. This class primarily consists of a list of `missing_variable`s plus slots containing metadata indicating how the `missing_variable`s relate to each other. Most operations that work for a `data.frame` also work for a missing_data.frame.

Usage

```
missing_data.frame(y, ...)
## Hidden arguments not included in the signature
## favor_ordered = TRUE, favor_positive = FALSE,
## subclass = NA_character_,
## include_missingness = TRUE, skip_correlation_check = FALSE
```

Arguments

y	Usually a data.frame, possibly a numeric matrix, possibly a list of missing_variables.
...	Hidden arguments. The favor_ordered and favor_positive arguments are passed to the missing_variable function and are documented under the type argument. Briefly, they affect the heuristics that are used to guess what class a variable should be coerced to. The subclass argument defaults to NA and can be used to specify that the resulting object should inherit from the missing_data.frame class rather than be an object of missing_data.frame class. Any further arguments are passed to the initialize-methods for a missing_data.frame. They currently are include_missingness, which defaults to TRUE and indicates that the missingness pattern of the other variables should be included when modeling a particular missing_variable, and skip_correlation_check, which defaults to FALSE and indicates whether to skip the default check for whether the observed values of each pair of missing_variables has a perfect absolute Spearman correlation.

Details

In most cases, the first step of an analysis is for a useR to call the missing_data.frame function on a data.frame whose variables have some NA values, which will call the missing_variable function on each column of the data.frame and return the list that fills the **variable** slot. The classes of the list elements will depend on the nature of the column of the data.frame and various fallible heuristics. The success rate can be enhanced by making sure that columns of the original data.frame that are intended to be categorical variables are (ordered if appropriate) factors with labels. Even in the best case scenario, it will often be necessary to utlize the change function to modify various discretionary aspects of the missing_variables in the **variables** slot of the missing_data.frame. The show method for a missing_data.frame should be utilized to get a quick overview of the missing_variables in a missing_data.frame and recognized what needs to be changed.

Value

The missing_data.frame constructor function returns an object of class missing_data.frame or that inherits from the missing_data.frame class.

Objects from the Class

Objects can be created by calls of the form new("missing_data.frame", ...). However, useRs almost always will pass a data.frame to the missing_data.frame constructor function to produce an object of missing_data.frame class.

Slots

This section is primarily aimed at developeRs. A missing_data.frame inherits from `data.frame` but has the following additional slots:

`variables`**:** Object of class `"list"` and each list element is an object that inherits from the `missing_variable-class`

`no_missing`**:** Object of class `"logical"`, which is a vector whose length is the same as the length of the **variables** slot indicating whether the corresponding `missing_variable` is fully observed

`patterns`**:** Object of class `factor` whose length is equal to the number of observation and whose elements indicate the missingness pattern for that observation

`DIM`**:** Object of class `"integer"` of length two indicating first the number of observations and second the length of the **variables** slot

`DIMNAMES`**:** Object of class `"list"` of length two providing the appropriate number rownames and column names

`postprocess`**:** Object of class `"function"` used to create additional variables from existing variables, such as interactions between two `missing_variable`s once their missing values have been imputed. Does not work at the moment

`index`**:** Object of class `"list"` whose length is equal to the number of `missing_variable`s with some missing values. Each list element is an integer vector indicating which columns of the **X** slot must be dropped when modeling the corresponding `missing_variable`

`X`**:** Object of `MatrixTypeThing-class` with rows equal to the number of observations and is loosely related to a `model.matrix`. Rather than repeatedly parsing a `formula` during the multiple imputation process, this **X** matrix is created once and some of its columns are dropped when modeling a `missing_variable` utilizing the **index** slot. The columns of the **X** matrix consists of numeric representations of the `missing_variable`s plus (by default) the unique missingness patterns

`weights`**:** Object of class `"list"` whose length is equal to one or the number of `missing_variable`s with some missing values. Each list element is passed to the corresponding argument of `bayesglm` and similar functions. In particular, some observations can be given a weight of zero, which should drop them when modeling some `missing_variable`s

`priors`**:** Object of class `"list"` whose length is equal to the number of `missing_variable`s and whose elements give appropriate values for the priors used by the model fitting function wraped by the `fit_model-methods`; see, e.g., `bayesglm`

`correlations`**:** Object of class `"matrix"` with rows and columns equal to the length of the **variables** slot. Its strict upper triangle contains Spearman `cor`relations between pairs of variables (ignoring missing values), and its strict lower triangle contains Squared Multiple Correlations (SMCs) between a variable and all other variables (ignoring missing values). If either a Spearman correlation or a SMC is very close to unity, there may be difficulty or error messages during the multiple imputation process.

`done`**:** Object of class `"logical"` of length one indicating whether the missing values have been imputed

`workpath`**:** Object of class `character` of length one indicating the path to a working directory that is used to store some objects

Methods

There are many methods that are defined for a missing_data.frame, although some are primarily intended for developers. The most relevant ones for users are:

change `signature(data = "missing_data.frame", y = "ANY", what = "character", to = "ANY")` which is used to change discretionary aspects of the `missing_variable`s in the **variables** slot of a missing_data.frame

hist `signature(x = "missing_data.frame")` which shows histograms of the observed variables that have missingness

image `signature(x = "missing_data.frame")` which plots an image of the **missingness** slot to visualize the pattern of missingness when `grayscale = FALSE` or the pattern of missingness in light of the observed values (`grayscale = TRUE`, the default)

mi `signature(y = "missing_data.frame", model = "missing")` which multiply imputes the missing values

show `signature(object = "missing_data.frame")` which gives an overview of the salient characteristics of the `missing_variables` in the **variables** slot of a missing_data.frame

summary `signature(object = "missing_data.frame")` which produces the same result as the `summary` method for a `data.frame`

There are also S3 methods for the `dim`, `dimnames`, and `names` generics, which allow functions like `nrow`, `ncol`, `rownames`, `colnames`, etc. to work as expected on `missing_data.frames`. Also, accessing and changing elements for a `missing_data.frame` mostly works the same way as for a `data.frame`

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`change`, `missing_variable`, `mi`, `experiment_missing_data.frame`, `multilevel_missing_data.frame`

Examples

```
# STEP 0: Get data
data(CHAIN, package = "mi")

# STEP 1: Convert to a missing_data.frame
mdf <- missing_data.frame(CHAIN) # warnings about missingness patterns
show(mdf)

# STEP 2: change things
mdf <- change(mdf, y = "log_virus", what = "transformation", to = "identity")

# STEP 3: look deeper
summary(mdf)
hist(mdf)
image(mdf)
```

```
# STEP 4: impute
## Not run:
imputations <- mi(mdf)

## End(Not run)

## An example with subsetting on a fully observed variable
data(nlsyV, package = "mi")
mdfs <- missing_data.frame(nlsyV, favor_positive = TRUE, favor_ordered = FALSE, by = "first")
mdfs <- change(mdfs, y = "momed", what = "type", to = "ord")
show(mdfs)
```

`03change` *Make Changes to Discretionary Characteristics of Missing Variables*

Description

These methods change the family, imputation method, size, type, and so forth of a `missing_variable`. They are typically called immediately before calling `mi` because they affect how the conditional expectation of each `missing_variable` is modeled.

Usage

```
change(data, y, to, what, ...)
change_family(data, y, to, ...)
change_imputation_method(data, y, to, ...)
change_link(data, y, to, ...)
change_model(data, y, to, ...)
change_size(data, y, to, ...)
change_transformation(data, y, to, ...)
change_type(data, y, to, ...)
```

Arguments

`data`	A `missing_data.frame` (typically) but can be missing for all but the change function
`y`	A character vector (typically) naming one or more `missing_variable`s within the `missing_data.frame` specified by the **data** argument. Alternatively, **y** can be the name of a class that inherits from `missing_variable`, in which case all `missing_variable`s of that class within `data` will be changed. Can also be an vector of integers or a logical vector indicating which `missing_variable`s to change.
`what`	Typically a character string naming what is to be changed, such as `"family"`, `"imputation_method"`, `"size"`, `"transformation"`, `"type"`, `"link"`, or `"model"`. Alternatively, it can be a scalar value, in which case all occurances of that value for the variable indicated by `y` will be changed to the value indicated by `to`

`to`	Typically a character string naming what `y` should be changed to, such as one of the admissible families, imputation methods, transformations, or types. If missing, then possible choices for the `to` argument will be helpfully printed on the screen. If `what` is a number, then `to` should be the number (or `NA`) that the value designated by `what` will be recoded to. See the Details section for more information.
`...`	Other arguments, not currently utilized

Details

In order to run `mi` correctly, data must first be specified to be ready for multiple imputation using the `missing_data.frame` function. For each variable, `missing_data.frame` will record information required by `mi`: the variable's type, distribution family, and link function; whether a variable should be standardized or tranformed by a log function or square root; what specific model to use for the conditional distribution of the variable in the `mi` algorithm and how to draw imputed values from this model; and whether additional rows (for the purposes of prediction) are required. `missing_data.frame` will attempt to guess the correct type, family, and link for each variable based on its class in a regular `data.frame`. These guesses can be checked with `show` and adjusted if necessary with `change`. Any further additions to the model in regards to variable transformations, custom conditional models, or extra non-observed predictive cases must be specified with `change` before `mi` is run.

In general, most users will only use the `change` command. `change` will then call `change_family`, `change_imputation_method`, `change_link`, `change_model`, `change_size`, `change_transformation`, or `change_type` depending on what characteristic is specified with the `what` option. The other change_* functions can be called directly but are primarily intended to be called indirectly by the change function.

`what = "type"` Change the subclass of variable(s) `y`. `to` should be a character vector whose elements are subclasses of the `missing_variable-class` and are documented further there. Among the most commonly used subclasses are `"unordered-categorical"`, `"ordered-categorical"`, `"binary"`, `"interval"`, `"continuous"`, `"count"`, and `"irrelevant"`.

`what = "family"` Change the distribution family for variable(s) `y`. `to` must be of class `family` or a list where each element is of class `family`. If a variable is of `binary-class`, then the family must be `binomial` (the default) or possibly `quasibinomial`. If a variable is of `ordered-categorical-class` or `unordered-categorical-class`, use the `multinomial` family. If a variable is of `count-class`, then the family must be `quasipoisson` (the default) or `poisson`. If a variable is continuous, there are more choices for its family, but `gaussian` is the default and the others are not supported yet.

`what = "link"` Change the link function for variable(s) `y`. `to` can be any of the supported link functions for the existing **family**. See `family` for details; however, not all of these link functions have appropriate `fit_model` and `mi-methods` yet.

`what = "model"` Change the conditional model for variable `y`. It usually is not necessary to change the model, since it is actually determined by the class, family, and link function of the variable. This option can be used, however, to employ models that are not among those listed above.`to` should be a character vector of length one indicating what model should be used during the imputation process. Valid choices for binary variables include `"logit"`, `"probit"` `"cauchit"`, `"cloglog"`, or quasilikelihoods `"qlogit"`, `"qprobit"`, `"qcauchit"`, `"qcloglog"`. For ordinal variables, valid choices include `"ologit"`, `"oprobit"`, `"ocauchit"`, and `"ocloglog"`.

For count variables, valid choices include `"qpoisson"` and `"poisson"`. Currently the only valid option for gaussian variables is `"linear"`. To change the model for unordered-categorical variables, see the estimator slot in `missing_variable`.

`what = "imputation_method"` Change the method for drawing imputed values from the conditional model specified for variable(s) `y`. `to` should be a character vector of length one or of the same length as `y` naming one of the following imputation methods: `"ppd"` (posterior predictive distribution), `"pmm"` (predictive mean matching), `"mean"` (mean imputation), `"median"` (median imputation), `"expectation"` (conditional expectation imputation).

`what = "size"` Optionally add additional rows for the purposes of prediction. `to` should be a single integer. If `to` is non-negative but less than the number of rows in the `missing_data.frame` given by the `data` argument, then `missing_data.frame` is augmented with `to` more rows, where all the additional observations are missing. If `to` is greater than the number of rows in the `missing_data.frame`given by the `data` argument, then the `missing_data.frame` is extended to have `to` rows, where the observations in the surplus rows are missing. If `to` is negative, then any additional rows in the `missing_data.frame` given by the `data` argument are removed to restore it to its original size.

`what = "transformation"` Specify a particular transformation to be applied to variable(s) `y`. `to` should be a character vector of length one or of the same length as `y` indicating what transformation function to use. Valid choices are `"identity"` for no transformation, `"standardize"` for standardization (using twice the standard deviation of the observed values), `"log"` for natural logarithm transformation, `"logshift"` for a `log(y + a)` transformation where `a` is a small constant, or `"sqrt"` for square-root transformation. Changing the transformation will also change the inverse transformation in the appropriate way. Any other value of `to` will produce an informative error message indicating that the transformation and inverse transformation need to be changed manually.

what = a value Finally, if both `what` and `to` are values then the former is recoded to the latter for all occurances within the missing variable indicated by `y`.

Value

If the **data** argument is not missing, then the method returns this argument with the specified changes. If **data** is missing, then the method returns an object that inherits from the `missing_variable-class` with the specified changes.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`, `missing_data.frame`

Examples

```
# STEP 0: GET DATA
data(nlsyV, package = "mi")
```

```
# STEP 1: CONVERT IT TO A missing_data.frame
mdf <- missing_data.frame(nlsyV)
show(mdf)

# STEP 2: CHANGE WHATEVER IS WRONG WITH IT
mdf <- change(mdf, y = "momrace", what = "type", to = "un")
mdf <- change(mdf, y = "income", what = "imputation_method", to = "pmm")
mdf <- change(mdf, y = "binary", what = "family", to = binomial(link = "probit"))
mdf <- change(mdf, y = 5, what = "transformation", to = "identity")
show(mdf)
```

04mi *Multiple Imputation*

Description

The mi function cannot be run in isolation. It is the most important step of a multi-step process to perform multiple imputation. The data must be specified as a missing_data.frame before mi is used to impute missing values for one or more missing_variables. An iterative algorithm is used where each missing_variable is modeled (using fit_model) as a function of all the other missing_variables and their missingness patterns. This documentation outlines the technical uses of the mi function. For a more general discussion of how to use mi for multiple imputation, see mi-package.

Usage

```
mi(y, model, ...)
## Hidden arguments:
## n.iter = 30, n.chains = 4, max.minutes = Inf, seed = NA, verbose = TRUE,
## save_models = FALSE, parallel = .Platform$OS.type != "windows"
```

Arguments

y	Typically an object that inherits from the missing_data.frame-class, although many methods are defined for subclasses of the missing_variable-class. Alternatively, y = "parallel" the appropriate parallel backend will be registered but no imputation performed. See the Details section.
model	Missing when y = "parallel" or when y inherits from the missing_data.frame-class but otherwise should be the result of a call to fit_model.
...	Further arguments, the most important of which are

n.iter number of iterations to perform, defaulting to 30

n.chains number of chains to use, ideally equal to the number of virtual cores available for use, and defaulting to 4

max.minutes hard time limit that defaults to 20

seed either NA, which is the default, or a psuedo-random number seed `verbose` logical scalar that is `TRUE` by default, indicating that progress of the iterative algorithm should be printed to the screen, which does not work under Windows when the chains are executed in parallel

`save_models` logical scalar that defaults to `FALSE` but if `TRUE` indicates that the models estimated on a frozen completed dataset should be saved. This option should be used if the user is interested in evaluating the quality of the models run after the last iteration of the `mi` algorithm, but saving these models consumes much more RAM

`debug` logical scalar indicating whether to run in debug mode, which forces the processing to be sequential, and allows developers to capture errors within chains

`parallel` if TRUE, then parallel processing is used, if available. If FALSE, sequential processing is used. In addition, ths argument may be an object produced by `makeCluster`

Details

It is important to distinguish the two `mi` methods that are most relevant to users from the many `mi` methods that are less relevant. The primary `mi` method is that where `y` inherits from the `missing_data.frame-class` and `model` is omitted. This method "does" the imputation according to the additional arguments described under ... above and returns an object of class `"mi"`. Executing two or more independent chains is important for monitoring the convergence of each chain, see `Rhats`.

If the chains have not converged in the amount of iterations or time specified, the second important `mi` method is that where `y` is an object of class `"mi"` and `model` is omitted, which continues a previous run of the iterative imputation algorithm. All the arguments described under ... above remain applicable, except for `n.chains` and `save_RAM` because these are established by the previous run that is being continued.

The numerous remaining methods are of less importance to users. One `mi` method is called when `y = "parallel"` and `model` is omitted. This method merely sets up the parallel backend so that the chains can be executed in parallel on the local machine. We use the `mclapply` function in the **parallel** package to implement parallel processing on non-Windows machines, and we use the **snow** package to implement parallel processing on Windows machines; we refer users to the documentation for these packages for more detail about parallel processing. Parallel processing is used by default on machines with multiple processors, but sequential processing can be used instead by using the `parallel=FALSE` option. If the user is not using a mulitcore computer, sequential processing is used instead of parallel processing.

The first two `mi` methods described above in turn call a `mi` method where `y` inherits from the `missing_data.frame-class` and `model` is that which is returned by one of the `fit_model-methods`. The methods impute values for the originally missing values of a `missing_variable` given a fitted model, according to the **imputation_method** slot of the `missing_variable` in question. Advanced users could define new subclasses of the `missing_variable-class` in which case it may be necessary to write such a `mi` method for the new class. It will almost certainly be necessary to add to the `fit_model-methods`. The existing `mi` and `fit-model-methods` should provide a template for doing so.

Value

If model is missing and n.chains is positive, then the mi method will return an object of class "mi", which has the following slots:

call the call to mi

data a list of missing_data.frames, one for each chain

total_iters an integer vector that records how many iterations have been performed

There are a few methods for such an object, such as show, summary, dimnames, nrow, ncol, etc.

If mi is called on a missing_data.frame with model missing and a nonpositive n.chains, then the missing_data.frame will be returned after allocating storeage.

If model is not missing, then the mi method will impute missing values for the y argument and return it.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

missing_data.frame, fit_model

Examples

```
# STEP 0: Get data
data(CHAIN, package = "mi")

# STEP 1: Convert to a missing_data.frame
mdf <- missing_data.frame(CHAIN) # warnings about missingness patterns
show(mdf)

# STEP 2: change things
mdf <- change(mdf, y = "log_virus", what = "transformation", to = "identity")

# STEP 3: look deeper
summary(mdf)

# STEP 4: impute
## Not run:
imputations <- mi(mdf)

## End(Not run)
```

`05Rhats` *Convergence Diagnostics*

Description

These functions are used to gauge whether `mi` has converged.

Usage

```
Rhats(imputations, statistic = c("moments", "imputations", "parameters"))
mi2BUGS(imputations, statistic = c("moments", "imputations", "parameters"))
```

Arguments

`imputations`	an object of `mi-class`
`statistic`	single character string among `"moments"`, `"imputations"`, and `"parameters"` indicating what statistic to monitor for convergence

Details

If `statistic = "moments"` (the default), then the mean and standard deviation of each variable will be monitored over the iterations. If `statistic = "imputations"`, then the imputed values will be monitored, which may be quite large and quite slow and is not possible if the `save_RAM = TRUE` flag was set in the call to the `mi` function. If `statistic = "parameters"`, then the estimated coefficients and ancillary parameters extracted by the `get_parameters-methods` will be monitored.

`Rhats` produces a vector of R-hat convergence statistics that compare the variance between chains to the variance across chains. Values closer to 1.0 indicate little is to be gained by running the chains longer, and in general, values greater than 1.1 indicate that the chains should be run longer. See Gelman, Carlin, Stern, and Rubin, "Bayesian Data Analysis", Second Edition, 2009, p.304 for more information about the R-hat statistic.

`mi2BUGS` outputs the history of the indicated statistic

Value

`mi2BUGS` returns an array while `Rhats` a vector of R-hat convergence statistics.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

Examples

```
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
  dim(mi2BUGS(imputations))
  Rhats(imputations)
```

`06pool` *Estimate a Model Pooling Over the Imputed Datasets*

Description

This function estimates a chosen model, taking into account the additional uncertainty that arises due to a finite number of imputations of the missing data.

Usage

```
pool(formula, data, m = NULL, FUN = NULL, ...)
```

Arguments

`formula`	a `formula` in the same syntax as used by `glm`
`data`	an object of `mi-class`
`m`	number of completed datasets to average over, which if `NULL` defaults to the number of chains used in `mi`
`FUN`	Function to estimate models or `NULL` which uses the same function as used in the `fit_model-methods` for the dependent variable
`...`	further arguments passed to `FUN`

Details

`FUN` is estimated on each of the `m` completed datasets according to the given `formula` and the results are combined using the Rubin Rules.

Value

An object of class `"pooled"` whose definition is subject to change but it has a `summary` and `display` method.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`mi`

Examples

```
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
analysis <- pool(ppvtr.36 ~ first + b.marr + income + momage + momed + momrace,
                 data = imputations)
display(analysis)
```

`07complete` *Extract the Completed Data*

Description

This function extracts several multiply imputed `data.frame`s from an object of `mi-class`.

Usage

```
complete(y, m, ...)
```

Arguments

`y`	An object of `mi-class` (typically) or `missing_data.frame-class` or `missing_variable-class`
`m`	If **y** is an object of `mi-class`, then `m` must be a specified integer indicating how many multiply imputed `data.frame`s to return or, if missing, the number of `data.frame`s will be equal to the length of the **data** slot in `y`. If `y` is not an object of `mi-class`, then **m** must be a specified integer indicating which iteration to use in the resulting `data.frame`, where any non-positive integer is a short hand for the last iteration.
`...`	Other arguments, not currently utilized

Details

Several functions within **mi** use `complete`, although the only reason in principle why a user should need to call `complete` is to create `data.frame`s to export to another program. For analysis, it is better to use the `pool` function, although currently `pool` might not offer all the necessary functionality.

Value

If **y** is an object of `mi-class` and `m` > 1, a `list` of `m` `data.frame`s is returned. Otherwise, a single `data.frame` is returned.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

mi-class

Examples

```
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
data.frames <- complete(imputations, 3)
lapply(data.frames, summary)
```

---

allcategorical_missing_data.frame

Class "allcategorical_missing_data.frame"

Description

This class inherits from the missing_data.frame-class but is customized for the situation where all the variables are categorical.

Details

The fit_model-methods for the allcategorical_missing_data.frame class implement a Gibbs sampler. However, it does not utilize any ordinal information that may be available. Continuous variables should be made into factors using the cut command before calling missing_data.frame.

Objects from the Class

Objects can be created by calls of the form new("allcategorical_missing_data.frame", ...). However, its users almost always will pass a data.frame to the missing_data.frame function and specify the subclass argument.

Slots

The allcategorical_missing_data.frame class inherits from the missing_data.frame-class and has three additional slots

Hstar Positive integer indicating the maximum number of latent classes

parameters A list that holds the current realization of the unknown parameters

latents An object of unordered-categorical-class that contains the current realization of the latent classes

Author(s)

Sophie Si for the algorithm and Ben Goodrich for the R implementation

See Also

`missing_data.frame`

Examples

```
rdf <- rdata.frame(n_full = 2, n_partial = 2,
                   restrictions = "stratified", types = "ord")
mdf <- missing_data.frame(rdf$obs, subclass = "allcategorical")
```

`bounded-continuous-class`
Class "bounded-continuous"

Description

The bounded-continuous class inherits from the `continuous-class` and is intended for variables whose observations fall within open intervals that have *known* boundaries. Although proportions satisfy this definition, the `proportion-class` should be used in that case. At the moment, a bounded continuous variable is modeled as if it were simply a continuous variable, but its `mi-methods` impute the missing values from a truncated normal distribution using the `rtruncnorm` function in the **truncnorm** package. Note that the default transformation is the identity so if another transformation is used, the bounds must be specified on the transformed data. Aside from these facts, the rest of the documentation here is primarily directed toward developers.

Objects from the Classes

Objects can be created that are of bounded-continuous class via the the `missing_variable` generic function by specifying `type = "bounded-continuous"` as well as `lower` and / or `upper`

Slots

The bounded-continuous class inherits from the continuous class and is intended for variables that are supported on a known interval. Its default transformation function is the identity transformation and its `imputation_method` must be `"ppd"`. It has two additional slots:

upper a numeric vector whose length is either one or the value of the `n_total` slot giving the upper bound for *every* observation; `NA`s are not allowed

lower a numeric vector whose length is either one or the value of the `n_total` slot giving the lower bound for *every* observation; `NA`s are not allowed

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`, `continuous-class`, `positive-continuous-class`, `proportion-class`

Examples

```
# STEP 0: GET DATA
data(CHAIN, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
lo_bound <- 0
hi_bound <- rep(Inf, nrow(CHAIN))
hi_bound[CHAIN$log_virus == 0] <- 6
log_virus <- missing_variable(ifelse(CHAIN$log_virus == 0, NA, CHAIN$log_virus),
                              type = "bounded-continuous", lower = lo_bound, upper = hi_bound)

show(log_virus)
```

`categorical` *Class "categorical" and Inherited Classes*

Description

The categorical class is a virtual class that inherits from the `missing_variable-class` and is the parent of the unordered-categorical and ordered-categorical classes. The ordered-categorical class is the parent of both the binary and interval classes. Aside from these facts, the rest of the documentation here is primarily directed toward developers.

Objects from the Classes

The categorical class is virtual, so no objects may be created from it. However, the `missing_variable` generic function can be used to instantiate an object that inherits from the categorical class by specifying `type = "unordered-categorical"`, `type = "ordered-categorical"`, `type = "binary"`, `type = "grouped-binary"`, or `type = "interval"`.

Slots

The unordered-categorical class inherits from the categorical class and has no additional slots but must have more than two uniquely observed values in its `raw_data` slot. The default `fit_model` method is a wrapper for the `multinom` function in the **nnet** package. The ordered-categorical class inherits from the categorical class and has one additional slot:

cutpoints Object of class `"numeric"` which is a vector of thresholds (sometimes estimated) that govern how an assumed latent variable is divided into observed ordered categories

The `fit_model` method for an ordered-categorical variable is, by default, a wrapper for `bayespolr`. The binary class inherits from the ordered-categorical class and has no additional slots. It must have exactly two uniquely observed values in its `raw_data` slot and its `fit_model` method is, by default, a wrapper for `bayespolr`. The grouped-binary class inherits from the binary class and has one additional slot:

strata Object of class `"character"` which is a vector (possibly of length one) of variable names that group the observations into strata. The named external variables should also be categorical.

The default `fit_model` method for a grouped-binary variable is a wrapper for the `clogit` function in the **survival** package and the variables named in the **strata** slot are passed to the `strata` function.

The interval class inherits from the ordered-categorical class, has no additional slots, and is intended for variables whose observed values are only known up to orderable intervals. Its `fit_model` method is, by default, a wrapper for `survreg` even though it may or may not be a "survival" model in any meaningful sense.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`

Examples

```
# STEP 0: GET DATA
data(nlsyV, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
momrace <- missing_variable(as.factor(nlsyV$momrace), type = "unordered-categorical")
show(momrace)
```

`censored-continuous-class` *The "censored-continuous" Class, the "truncated-continuous" Class and Inherited Classes*

Description

The censored-continuous class and the truncated-continuous class are both virtual and both inherit from the `continuous-class` and each is the parent of four classes that differ depending on whether the lower and upper bounds are numeric vectors or functions. A censored observation is one whose exact value is not observed. A truncated observation is one whose exact value is not observed and which implies that values on some *other* variables are not observed for that unit of observation. An example of truncation might be that some taxation forms are not required when a person's income falls below a certain threshold. The methods for these classes are not working yet. Aside from these facts, the rest of the documentation here is primarily directed toward developeRs.

Objects from the Classes

Both the censored-continuous class and the truncated-continuous class are virtual, so no objects can be created with these classes. However, the `missing_variable` generic function can be used to create an object that inherits from one of their subclasses by specifying `type = "NNcensored-continuous"`, `type = "NFcensored-continuous"`, `type = "FNcensored-continuous"`, `type = "FFcensored-continuous"`, `type = "NNtruncated-continuous"`, `type = "NFtruncated-continuous"`, `type = "FNtruncated-continuous"`, type = "FFtruncated-continuous". When doing so, the lower and upper slots need to be specified appropriately.

Slots

The censored-continuous class and the truncated-continuous class are both virtual, both inherit from the continuous class, both use the identity transformation by default, and both have two additional slots:

upper The upper bound for each observation

lower The lower bound for each observation

Both the censored-continuous class and the truncated-continuous class have four subclasses that differ depending on whether the upper and / or lower bounds are numeric vectors or functions that output numeric vectors (scalars are recycled and can be Inf). These subclasses are

NN_censored-continuous where both the lower and upper bounds are numeric vectors

FN_censored-continuous where the lower bound is a function and the upper bound is a numeric vector

NF_censored-continuous where the lower bound is a numeric vector and the upper bound is a function

FF_censored-continuous where both the lower and upper bounds are functions

NN_truncated-continuous where both the lower and upper bounds are numeric vectors

FN_truncated-continuous where the lower bound is a function and the upper bound is a numeric vector

NF_truncated-continuous where the lower bound is a numeric vector and the upper bound is a function

FF_truncated-continuous where both the lower and upper bounds are functions

Author(s)

Ben Goodrich, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

missing_variable, continuous-class

Examples

```
# STEP 0: GET DATA
data(CHAIN, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
#log_virus <- missing_variable(CHAIN$log_virus, type = "NN_censored-continuous",
#                              lower = 0, upper = Inf)
#show(log_virus)
```

CHAIN *Subset of variables from the CHAIN project*

Description

The CHAIN project was a longitudinal cohort study of people living with HIV in New York City, which was recruited in 1994 from a large number of medical care and social service agencies serving HIV in New York City. This subset of data pertain to the sixth round of interviews.

Usage

```
data(CHAIN)
```

Format

A `data.frame` with 532 observations on the following 8 variables.

`log_virus` log of self reported viral load level, where zero represents an undetectable level.

`age` age at time of the interview

`income` annual family income in 10 intervals

`healthy` a continuous scale of physical health with a theoretical range between 0 and 100 where better health is associated with higher scale values

`mental` a binary measure of poor mental health (1=Yes, 0=No)

`damage` ordered interval for the CD4 count, which is an indicator of how much damage HIV has caused to the immune system

`treatment` a three-level ordered variable: 0=Not currently taking HAART (Highly Active AntiretRoviral Therapy) 1=taking HAART but nonadherent, 2=taking HAART and adherent

Details

A missing value in the log virus load level was assigned to individuals who either could not recall their viral load level, did not have a viral load test in the six month preceding the interview, or reported their viral loads as "good" or "bad".

Source

http://cchps.columbia.edu/research.cfm

References

Messeri P, Lee G, Abramson DA, Aidala A, Chiasson MA, Jones JD. (2003). “Antiretroviral therapy and declining AIDS mortality in New York City”. *Medical Care* 41:512–521.

`continuous` *Class "continuous"*

Description

The continuous class inherits from the `missing_variable-class` and is the parent of the following classes: `semi-continuous`, `censored-continuous`, `truncated-continuous`, and `bounded-continuous`. The distinctions among these subclasses are given on their respective help pages. Aside from these facts, the rest of the documentation here is primarily directed toward developers.

Objects from the Classes

Objects can be created that are of class continuous via the `missing_variable` generic function by specifying `type = "continuous"`

Slots

The continuous class inherits from the `missing_variable` class and has the following additional slots:

transformation Object of class `"function"` which is passed the `raw_data` slot and whose returned value is assigned to the `data` slot. By default, this function is the "standardize" transformation, using the mean and *twice* the standard deviation of the observed values

inverse_transformation Object of class `"function"` which is the inverse of the function in the `transformation` slot.

transformed Object of class `"logical"` of length one indicating whether the `data` slot is in the "transformed" state or the "untransformed" state

known_transformations Object of class `"character"` indicating which transformations are possible for this variable

The `fit_model` method for a continuous variable is, by default, a wrapper for `bayesglm` and its `family` slot is, by default, `gaussian`

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`, `semi-continuous-class`, `censored-continuous-class`, `truncated-continuous-class`, `bounded-continuous-class`

Examples

```
# STEP 0: GET DATA
data(nlsyV, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
income <- missing_variable(nlsyV$income, type = "continuous")
show(income)
```

`count-class` *Class "count"*

Description

The count class inherits from the `missing_variable-class` and is intended for count data. Aside from these facts, the rest of the documentation here is primarily directed toward developers.

Objects from the Classes

Objects can be created that are of count class via the `missing_variable` generic function by specifying `type = "count"`

Slots

The count class inherits from the missing_variable class and its `raw_data` slot must consist of nonnegative integers. Its default family is `quasipoisson` and its default `fit_model` method is a wrapper for `bayesglm`. The other possibility for the family is `poisson` but is not recommended due to its overly-restrictive nature.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`, `continuous-class`, `positive-continuous-class`, `proportion-class`

Examples

```
# STEP 0: GET DATA
data(CHAIN, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
age <- missing_variable(as.integer(CHAIN$age), type = "count")
show(age)
```

experiment_missing_data.frame
Class "experiment_missing_data.frame"

Description

This class inherits from the missing_data.frame-class but is customized for the situation where the sample is a randomized experiment.

Details

The fit_model-methods for the experiment_missing_data.frame class take into account the special nature of a randomized experiment. At the moment, the treatment variable must be binary and fully observed.

Objects from the Class

Objects can be created by calls of the form new("experiment_missing_data.frame", ...). However, its users almost always will pass a data.frame to the missing_data.frame function and specify the subclass and concept arguments.

Slots

The experiment_missing_data.frame class inherits from the missing_data.frame-class and has two additional slots

concept Object of class factor whose length is equal to the number of variables and whose levels are "treatment", "covariate" and "outcome"

case Object of class character of length one, indicating whether the missingness is in the outcomes only, in the covariates only, or in both the outcomes and covariates. This slot is filled automatically by the initialize method

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

missing_data.frame

Examples

```
rdf <- rdata.frame(n_full = 2, n_partial = 2,
                   restrictions = "stratified", experiment = TRUE,
                   types = c("t", "ord", "con", "pos"),
                   treatment_cor = c(0, 0, NA, 0, NA))
Sigma <- tcrossprod(rdf$L)
```

```
rownames(Sigma) <- colnames(Sigma) <- c("treatment", "X_2", "y_1", "Y_2",
                                        "missing_y_1", "missing_Y_2")
print(round(Sigma, 3))

concept <- as.factor(c("treatment", "covariate", "covariate", "outcome"))
mdf <- missing_data.frame(rdf$obs, subclass = "experiment", concept = concept)
```

fit_model *Wrappers To Fit a Model*

Description

The methods are called by the mi function to model a given missing_variable as a function of all the other missing_variables and also their missingness pattern. By overwriting these methods, users can change the way a missing_variable is modeled for the purposes of imputing its missing values. See also the table in missing_variable.

Usage

```
fit_model(y, data, ...)
```

Arguments

y	An object that inherits from missing_variable-class or missing
data	A missing_data.frame
...	Additional arguments, not currently utilized

Details

In mi, each missing_variable is modeled as a function of all the other missing_variables plus their missingness pattern. The fit_model methods are typically short wrappers around a statistical model fitting function and return the estimated model. The model is then passed to one of the mi-methods to impute the missing values of that missing_variable.

Users can easily overwrite these methods to estimate a different model, such as wrapping glm instead of bayesglm. See the source code for examples, but the basic outline is to first extract the X slot of the missing_data.frame, then drop some of its columns using the index slot of the missing_data.frame, next pass the result along with the data slot of y to a statistical fitting function, and finally returned the appropriately classed result (along with the subset of X used in the model).

Many of the optional arguments to a statistical fitting function can be specified using the slots of y (e.g. its family slot) or the slots of **data** (e.g. its weights slot).

The exception is the method where y is missing, which is used internally by mi, and should *not* be overwritten unless great care is taken to understand its role.

Value

If y is missing, then the modified missing_data.frame passed to data is returned. Otherwise, the estimated model is returned as a classed list object.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

missing_variable, mi, get_parameters

Examples

```
getMethod("fit_model", signature(y = "binary", data = "missing_data.frame"))
setMethod("fit_model", signature(y = "binary", data = "missing_data.frame"), def =
function(y, data, ...) {
  to_drop <- data@index[[y@variable_name]]
  X <- data@X[, -to_drop]
  start <- NULL
  # using glm.fit() instead of bayesglm.fit()
  out <- glm.fit(X, y@data, weights = data@weights[[y@variable_name]], start = start,
                 family = y@family, Warning = FALSE, ...)
  out$x <- X
  class(out) <- c("glm", "lm") # not "bayesglm" class anymore
  return(out)
})
## Not run:
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
imputations <- mi(imputations) # will use new fit_model() method for binary variables

## End(Not run)
```

---

get_parameters *An Extractor Function for Model Parameters*

Description

This function is not intended to be called directly by users. During the multiple imputation process, the mi function estimates models and stores the estimated parameters in the parameters slot of an object that inherits from the missing_variable-class. The get_parameter function simply extracts these parameters for storeage, which are usually the estimated coefficients but may also include ancillary parameters.

Usage

```
get_parameters(object, ...)
```

Arguments

object	Usually an estimated model, such as that produced by glm
...	Additional arguments, currently not used

Details

There is method for the object produced by polr, which also returns the estimated cutpoints in a proportional odds model. However, the default method simply calls coef and returns the result. If users implement their own models, it may be necessary to write a short `get_parameters` method.

Value

A numeric vector of estimated parameters

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

fit_model

Examples

```
showMethods("get_parameters")
```

---

hist *Histograms of Multiply Imputed Data*

Description

This function creates a histogram from an object of missing_data.frame-class or mi-class

Usage

```
hist(x, ...)
```

Arguments

x	an object of missing_data.frame-class or mi-class
...	further arguments passed to plot.histogram

Details

When called on an object of missing_data.frame-class, the histograms of the observed data are generated, one for each missing_variable but grouped on a single page. When called on an object of mi-class, the histograms of the observed, imputed, and completed data are generated, one for each missing_variable, grouped on a single page for each chain.

Value

An invisible `NULL` is returned with a side-effect of creating a plot

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

hist

Examples

```
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
hist(imputations)
```

---

irrelevant — *Class "irrelevant" and Inherited Classes*

Description

The irrelevant class inherits from the missing_variable-class and is used to designate variables that are excluded from the models used to impute the missing values of “relevant” variables. For example, if a survey has an “id” variable that simply distinguishes observations, the user should designate it as irrelevant, although it will automatically be classified so if its name is either “id” or starts with punctuation (including underscores). The fixed class inherits from the irrelevant class and is used for variables that are constant (within a sample). A variable that is instantiated from the fixed class cannot have any missing values. The group class inherits from the fixed class and is used like a factor to spit samples in multilevel modeling; see multilevel_missing_data.frame-class. None of these classes have an additional slots. Aside from these facts, the rest of the documentation here is primarily directed toward developeRs.

Objects from the Classes

The missing_variable generic function can be used to instantiate an object that inherits from the irrelevant class by specifying `type = "irrelevant"`, `type = "fixed"`, or `type = "group"`.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

missing_variable-class

Examples

```
# STEP 0: GET DATA
data(nlsyV, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
first <- missing_variable(as.factor(nlsyV$first), type = "group")
show(first)
```

---

mi2stata *Exports completed data in Stata (.dta) or comma-separated (.csv) format*

Description

This function exports completed data from an object of `mi-class` in which `m` completed `data.frame`s are appended to the end of the raw data. Two additional variables are added which indicate the row number and distinguish the `data.frame`s. The outputed file is either Stata (.dta) or comma-separated (.csv) format, and can be easily registered in Stata as multiply imputed data.

Usage

```
mi2stata(imputations, m, file, missing.ind=FALSE, ...)
```

Arguments

`imputations`	Object of `mi-class`
`m`	The number of completed datasets to append onto the raw data
`file`	The filename, either a full path or relative to the working directory, where the file will be saved. Filenames must end in either '.dta' or '.csv'. Files with names ending in '.dta' will be saved as a Stata data file, and files with names ending in '.csv' will be saved as a comma-separated file.
`missing.ind`	If `TRUE`, includes a binary variable for each variable with `NA` values, indicating the observations which were originally missing. Defaults to `FALSE`.
`...`	Further arguments passed to `write.dta` for Stata files, or to `write.table` for .csv files.

Details

The function calls `complete` to construct `m` completed `data.frame`s, and uses `rbind` to append them to the bottom of the raw data that still contains all of the missing values. Two new variables are added: `_mi`, which contains the observation numbers; and `_mj`, which indexes the `data.frame`s.

To save a Stata .dta file, end the filename with '.dta'. To save a comma-separated file, end the filename with .csv'. Stata files are loaded into Stata using Stata's `use` command, and comma-separated files can be loaded by typing `insheet using` *filename*`, comma names clear`. Once the file is loaded into Stata, the data must be registered as multiply imputed before any subsequent analyses can be performed. In Stata version 11 or later, type `mi import mice` to register the data.

The `_mi` and `_mj` variables will be replaced by variables named `_mi_id` and `_mi_m` respectively. In Stata version 10 or earlier, install the `MIM` package by typing `findit mim` and installing package `st0139_1`. Then the prefix `mim:` must be added to any command using the multiply imputed data.

Any observations which are unpossible (legitimately skipped, and are not imputed, see `missing_variable`) will remain missing in the complete data, but will not be indicated as missing by these variables. If there are any unpossible values, missing indicators are included automatically.

Value

`NULL`

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`complete`, `mi`, `write.dta`, `write.table`

Examples

```
fn <- paste(tempfile(), "dta", sep = ".")
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
mi2stata(imputations, m=5, file=fn , missing.ind=TRUE)
```

`mipply` *Apply a Function to a Object of Class mi*

Description

This function is a wrapper around `sapply` that is invoked on the `data` slot of an object of `mi-class` and / or on an object of `missing_data.frame-class` after being coerced to a `data.frame`

Usage

```
mipply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE,
       columnwise = TRUE, to.matrix = FALSE)
```

Arguments

`X`	Object of `mi-class`, `missing_data.frame-class`, `missing_variable-class`, `mi_list-class`, or `mdf_list-class`
`FUN`	Function to call
`...`	Further arguments passed to `FUN`, currently broken

`simplify`	If `TRUE`, coerces result to a vector or matrix if possible
`USE.NAMES`	ignored but included for compatibility with `sapply`
`columnwise`	logical indicating whether to invoke `FUN` on the columns of a `missing_data.frame` after coercing it to a `data.frame` or a `matrix` or to invoke `FUN` on the “whole” `data.frame` or `matrix`
`to.matrix`	Logical indicating whether to coerce each `missing_data.frame` to a numeric `matrix` or to a `data.frame`. The default is `FALSE`, in which case the `data.frame` will include `factor`s if any of the `missing_variable`s inherit from `categorical-class`

Details

The `columnwise` and `to.matrix` arguments are the only additions to the argument list in `sapply`, see the Examples section for an illustration of their use. Note that functions such as `mean` only accept `numeric` inputs, which can produce errors or warnings when `to.matrix` = `FALSE`.

Value

A list, vector, or matrix depending on the arguments

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`sapply`

Examples

```
if(!exists("imputations", env = .GlobalEnv)) {
  imputations <- mi:::imputations # cached from example("mi-package")
}
round(mipply(imputations, mean, to.matrix = TRUE), 3)
mipply(imputations, summary, columnwise = FALSE)
```

`multilevel_missing_data.frame`
Class "multilevel_missing_data.frame"

Description

This class inherits from the `missing_data.frame-class` but is customized for the situation where the sample has a multilevel structure.

Details

The `fit_model-methods` for the multilevel_missing_data.frame class will, by default, utilize multilevel modeling techniques that shrink the estimated parameters for each group toward their global means.

Objects from the Class

Objects can be created by calls of the form `new("multilevel_missing_data.frame", ...)`. However, its users almost always will pass a `data.frame` to the `missing_data.frame` function and specify the `subclass` and `groups` arguments.

Slots

The multilevel_missing_data.frame class inherits from the `missing_data.frame-class` and has two additional slots

groups Object of class `character` indicating which variables define the multilevel structure

mdf_list Object of class `mdf_list` whose elements contain a `missing_data.frame` for each group. This slot is filled automatically by the `initialize` method.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_data.frame`

Examples

```
## Write example
```

`multinomial` *The multinomial family*

Description

This function is a returns a `family` and is a generalization of `binomial`. users would only need to call it when calling `change` with `what = "family", to = multinomial(link = 'logit')`

Usage

```
multinomial(link = "logit")
```

Arguments

`link`	character string among those supported by `binomial`

Details

This function is mostly cosmetic. The `family` slot for an object of `unordered-categorical-class` must be `multinomial(link = 'logit')`. For an object of `ordered-categorical-class` but not its subclasses, the `family` slot must be `multinomial()` but the link function can differ from its default `("logit")`

Value

A `family` object

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`family`, `binomial`

Examples

```
multinomial()
```

`nlsyV` *National Longitudinal Survey of Youth Extract*

Description

This dataset pertains to children and their families in the United States and is intended to illustrate missing data issues. Note that although the original data are longitudinal, this extract is not.

Usage

```
data(nlsyV)
```

Format

A data frame with 400 randomly subsampled observations on the following 7 variables.

`ppvtr.36` a numeric vector with data on the Peabody Picture Vocabulary Test (Revised) administered at 36 months

`first` indicator for whether child was first-born

`b.marr` indicator for whether mother was married when child was born

`income` a numeric vector with data on family income in year after the child was born

`momage` a numeric vector with data on the age of the mother when the child was born

`momed` educational status of mother when child was born (1 = less than high school, 2 = high school graduate, 3 = some college, 4 = college graduate) `momrace` race of mother (1 = black, 2 = Hispanic, 3 = white)

Note that **momed** would typically be an ordered `factor` while **momrace** would typically be an unorderd `factor` but both are `numeric` in this `data.frame` in order to illustrate the mechanism to `change` the type of a `missing_variable`

Source

National Longitudinal Survey of Youth, 1997, https://www.bls.gov/nls/nlsy97.htm

Examples

```
data(nlsyV)
summary(nlsyV)
```

`positive-continuous-class`

Class "positive-continuous" and Inherited Classes

Description

The positive-continuous class inherits from the `continuous-class` and is the parent of the proportion class. In both cases, no observations can be zero, and in the case of the proportion class, no observations can be one. The `nonnegative-continuous-class` and the `SC_proportion-class` are appropriate for those situations. Aside from these facts, the rest of the documentation here is primarily directed toward developeRs.

Objects from the Classes

Objects can be created that are of positive-continuous or proportion class via the `missing_variable` generic function by specifying `type = "positive-continuous"` or `type = "proportion"`

Slots

The default transformation for the positive-continuous class is the `log` function. The proportion class inherits from the positive-continuous class and has the identity transformation and the `binomial` family as defaults, in which case the `fit_model-methods` call the `betareg` function in the **betareg** package. Alternatively, the transformation could be an inverse CDF like the `qnorm` function and the family could be `gaussian`, in which case the `fit_model-methods` call the `bayesglm` function in the **arm** package.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`, `continuous-class`, `positive-continuous-class`, `proportion-class`

Examples

```
# STEP 0: GET DATA
data(CHAIN, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
healthy <- missing_variable(CHAIN$healthy / 100, type = "proportion")
show(healthy)
```

`rdata.frame` *Generate a random data.frame with tunable characteristics*

Description

This function generates a random `data.frame` with a missingness mechanism that is used to impose a missingness pattern. The primary purpose of this function is for use in simulations

Usage

```
rdata.frame(N = 1000,
          restrictions = c("none", "MARish", "triangular", "stratified", "MCAR"),
             last_CPC = NA_real_, strong = FALSE, pr_miss = .25, Sigma = NULL,
             alpha = NULL, experiment = FALSE,
             treatment_cor = c(rep(0, n_full - 1), rep(NA, 2 * n_partial)),
             n_full = 1, n_partial = 1, n_cat = NULL,
             eta = 1, df = Inf, types = "continuous", estimate_CPCs = TRUE)
```

Arguments

`N`	integer indicating the number of observations
`restrictions`	character string indicating what restrictions to impose on the the missing data mechansim, see the Details section
`last_CPC`	a numeric scalar between -1 and 1 exclusive or `NA_real_` (the default). If not `NA_real_`, then this value will be used to construct the correlation matrix from which the data are drawn. This option is useful if restrictions is `"triangular"` or `"stratified"`, in which case the degree to which `last_CPC` is not zero causes a violation of the Missing-At-Random assumption that is confined to the last of the partially observed variables
`strong`	Integer among 0, 1, and 2 indicating how strong to make the instruments with multiple partially observed variables, in which case the missingness indicators for each partially observed variable can be used as instruments when predicting missingness on other partially observed variables. Only applies when `restrictions = "triangular"`
`pr_miss`	numeric scalar on the (0,1) interval or vector of length `n_partial` indicating the proportion of observations that are missing on partially observed variables

`Sigma`	Either `NULL` (the default) or a correlation matrix of appropriate order for the variables (including the missingness indicators). By default, such a matrix is generated at random.
`alpha`	Either `NULL`, `NA`, or a numeric vector of appropriate length that governs the skew of a multivariate skewed normal distribution; see `rmsn`. The appropriate length is `n_full - 1 + 2 * n_partial` iff none of the variable types is nominal. If some of the variable types are nominal, then the appropriate length is `n_full - 1 + 2 * n_partial + sum(n_cat) - length(n_cat)`. If `NULL`, `alpha` is taken to be zero, in which case the data-generating process has no skew. If `NA`, `alpha` is drawn from `rt` with `df` degrees of freedom
`experiment`	logical indicating whether to simulate a randomized experiment
`treatment_cor`	Numeric vector of appropriate length indicating the correlations between the treatment variable and the other variables, which is only relevant if `experiment = TRUE`. The appropriate length is `n_full - 1 + 2 * n_partial` iff none of the variable types is nominal. If some of the variable types are nominal, then the appropriate length is `n_full - 1 + 2 * n_partial + sum(n_cat) - length(n_cat)`. If treatment_cor is of length one and is zero, then it will be recylced to the appropriate length. The treatment variable should be uncorrelated with intended covariates and uncorrelated with missingness on intended covariates. If any elements of treatment_cor are `NA`, then those elements will be replaced with random draws. Note that the order of the random variables is: all fully observed variables,all partially observed but not nominal variables, all partially observed nominal variables, all missingness indicators for partially observed variables.
`n_full`	integer indicating the number of fully observed variables
`n_partial`	integer indicating the number of partially observed variables
`n_cat`	Either `NULL` or an integer vector (possibly of length one) indicating the number of categories in each partially observed nominal or ordinal variable; see the Details section
`eta`	Positive numeric scalar which serves as a hyperparameter in the data-generating process. The default value of 1 implies that the correlation matrix among the variables is jointly uniformally distributed, using essentially the same logic as in the **clusterGeneration** package
`df`	positive numeric scalar indicating the degress of freedom for the (possibly skewed) multivariate t distribution, which defaults to `Inf` implying a (possibly skewed) multivariate normal distribution
`types`	a character vector (possibly of length one, in which case it is recycled) indicating the type for each fully observed and partially observed variable, which currently can be among `"continuous"`, `"count"`, `"binary"`, `"treatment"` (which is binary), `"ordinal"`, `"nominal"`, `"proportion"`, `"positive"`. See the Details section. Unique abbreviations are acceptable.
`estimate_CPCs`	A logical indicating whether the canonical partial correlations between the partially observed variables and the latent missingnesses should be estimated. The default is `TRUE` but considerable wall time can be saved by switching it to `FALSE` when there are many partially observed variables.

Details

By default, the correlation matrix among the variables and missingness indicators is intended to be close to uniform, although it is often not possible to achieve exactly. If `restrictions = "none"`, the data will be Not Missing At Random (NMAR). If `restrictions = "MARish"`, the departure from Missing At Random (MAR) will be minimized via a call to `optim`, but generally will not fully achieve MAR. If `restrictions = "triangular"`, the MAR assumption will hold but the missingness of each partially observed variable will only depend on the fully observed variables and the other latent missingness indicators. If `restrictions = "stratified"`, the MAR assumption will hold but the missingness of each partially observed variable will only depend on the fully observed variables. If `restrictions = "MCAR"`, the Missing Completely At Random (MCAR) assumption holds, which is much more restrictive than MAR.

There are some rules to follow, particularly when specifying `types`. First, if `experiment = TRUE`, there must be exactly one treatment variable (taken to be binary) and it must come first to ensure that the elements of `treatment_cor` are handled properly. Second, if there are any partially observed nominal variables, they must come last; this is to ensure that they are conditionally uncorrelated with each other. Third, fully observed nominal variables are not supported, but they can be made into ordinal variables and then converted to nominal after the fact. Fourth, including both ordinal and nominal partially observed variables is not supported yet, Finally, if any variable is specified as a count, it will not be exactly consistent with the data-generating process. Essentially, a count variable is constructed from a continuous variable by evaluating `pt` on it and passing that to `qpois` with an intensity parameter of 5. The other non-continuous variables are constructed via some transformation or discretization of a continuous variable.

If some partially observed variables are either ordinal or nominal (but not both), then the `n_cat` argument governs how many categories there are. If `n_cat` is `NULL`, then the number of categories defaults to three. If `n_cat` has length one, then that number of categories will be used for all categorical variables but must be greater than two. Otherwise, the length of `n_cat` must match the number of partially observed categorical variables and the number of categories for the ith such variable will be the ith element of `n_cat`.

Value

A list with the following elements:

1. true a `data.frame` containing no `NA` values
2. obs a `data.frame` derived from the previous with some `NA` values that represents a dataset that could be observed
3. empirical_CPCs a numeric vector of empirical Canonical Partial Correlations, which should differ only randomly from zero iff `MAR = TRUE` and the data-generating process is multivariate normal
4. L a Cholesky factor of the correlation matrix used to generate the true data

In addition, if `alpha` is not `NULL`, then the following elements are also included:

1. alpha the `alpha` vector utilized
2. sn_skewness the skewness of the multivariate skewed normal distribution in the population; note that this value is only an approximation of the skewness when `df < Inf`
3. sn_kurtosis the kurtosis of the multivariate skewed normal distribution in the population; note that this value is only an approximation of the kurtosis when `df < Inf`

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`data.frame`, `missing_data.frame`

Examples

```
rdf <- rdata.frame(n_partial = 2, df = 5, alpha = rnorm(5))
print(rdf$empirical_CPCs) # not zero
rdf <- rdata.frame(n_partial = 2, restrictions = "triangular", alpha = NA)
print(rdf$empirical_CPCs) # only randomly different from zero
print(rdf$L == 0) # some are exactly zero by construction
mdf <- missing_data.frame(rdf$obs)
show(mdf)
hist(mdf)
image(mdf)
# a randomized experiment
rdf <- rdata.frame(n_full = 2, n_partial = 2,
                   restrictions = "triangular", experiment = TRUE,
                   types = c("t", "ord", "con", "pos"),
                   treatment_cor = c(0, 0, NA, 0, NA))
Sigma <- tcrossprod(rdf$L)
rownames(Sigma) <- colnames(Sigma) <- c("treatment", "X_2", "y_1", "Y_2",
                                        "missing_y_1", "missing_Y_2")
print(round(Sigma, 3))
```

semi-continuous-class *Class "semi-continuous" and Inherited Classes*

Description

The `semi-continuous` class inherits from the `continuous-class` and is the parent of the `nonnegative-continuous` class, which in turn is the parent of the `SC_proportion` class for semi-continuous variables. A semi-continuous variable has support on one or more point masses and a continuous interval. The `semi-continuous` class differs from the `censored-continuous-class` and the `truncated-continuous-class` in that observations that fall on the point masses are bonafide data, rather than indicators of censoring or truncation. If there are no observations that fall on a point mass, then either the `continuous-class` or one of its other subclasses should be used. Aside from these facts, the rest of the documentation here is primarily directed toward developers.

Objects from the Classes

Objects can be created that are of `semi-continuous`, `nonnegative-continuous`, or `SC_proportion` class via the `missing_variable` generic function by specifying `type = "semi-continuous" type = "nonnegative-continuous"`, `type = "SC_proportion"`.

Slots

The semi-continuous class inherits from the continuous class and is intended for variables that, for example have a point mass at certain points and are continuous in between. Thus, its default transformation is the identity transformation, which is to say no transformation in practice. It has one additional slot.

indicator Object of class `"ordered-categorical"` that indicates whether an observed value falls on a point mass or the continuous interval in between. By convention, zero signifies an observation that falls within the continuous interval

At the moment, there are no methods for the semi-continuous class. However, the basic approach to modeling a semi-continuous variable has two steps. First, the **indicator** is modeled using the methods that are defined for it and its missing values are imputed. Second, the continuous part of the semi-continuous variable is modeled using the same techniques that are used when modeling continuous variables. Note that in the second step, only a subset of the observations are modeled, although this subset possibly includes values that were originally missing in which case they are imputed.

The nonnegative-continuous class inherits from the semi-continuous class, which has its point mass at zero and is continuous over the positive real line. By default, the transformation for the positive part of a nonnegative-continuuos variable is `log(y + a)`, where `a` is a small constant determined by the observed data. If a variable is strictly positive, the `positive-continuous-class` should be used instead.

The SC_proportion class inherits from the nonnegative-continuous class. It has no additional slots, and the only supported transformation function is the `(y * (n - 1) + .5) / n` function. Its default `fit_model` method is a wrapper for the `betareg` function in the **betareg** package. Its **family** must be `binomial` so that its `link` function can be passed to `betareg` If the observed values fall strictly on the open unit interval, the `proportion-class` should be used instead.

Author(s)

Ben Goodrich and Jonathan Kropko, for this version, based on earlier versions written by Yu-Sung Su, Masanao Yajima, Maria Grazia Pittau, Jennifer Hill, and Andrew Gelman.

See Also

`missing_variable`, `continuous-class`, `positive-continuous-class`, `proportion-class`

Examples

```
# STEP 0: GET DATA
data(nlsyV, package = "mi")

# STEP 0.5 CREATE A missing_variable (you never need to actually do this)
income <- missing_variable(nlsyV$income, type = "nonnegative-continuous")
show(income)
```

Index