

# Package ‘rMIDAS’

September 29, 2020

**Title** Multiple Imputation using Denoising Autoencoders

**Version** 0.1.0

## **Description**

A tool that allows users to impute missing data with 'MIDAS', a multiple imputation method using denoising autoencoders as documented in Lall and Robinson (2020) <doi:10.33774/apsa-2020-3tk40-v3>. This method has significant accuracy and efficiency advantages over other multiple imputation strategies, particularly when run on large datasets with many columns or categories. Alongside interfacing with 'Python' to run the core algorithm, this package contains tools to process the data before and after model training, run imputation model diagnostics, generate multiple completed datasets, and estimate multiply-imputed regression models.

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 7.1.1

**Depends** R (>= 3.6.0), data.table, mltools, reticulate

## **Imports**

**Suggests** testthat

**License** Apache License (>= 2.0)

**URL** <https://github.com/MIDASverse/rMIDAS>

**BugReports** <https://github.com/MIDASverse/rMIDAS/issues>

**NeedsCompilation** no

**Author** Thomas Robinson [aut, cre, cph]  
(<<https://orcid.org/0000-0001-7097-1599>>),  
Ranjit Lall [aut, cph] (<<https://orcid.org/0000-0003-1455-3506>>),  
Alex Stenlake [ctb, cph]

**Maintainer** Thomas Robinson <[ts.robinson1994@gmail.com](mailto:ts.robinson1994@gmail.com)>

**Repository** CRAN

**Date/Publication** 2020-09-29 08:10:03 UTC

R topics documented:

add_bin_labels . . . . .	2
coalesce_one_hot . . . . .	3
col_minmax . . . . .	3
combine . . . . .	4
complete . . . . .	4
convert . . . . .	6
import_midas . . . . .	7
mid_py_setup . . . . .	8
na_to_nan . . . . .	8
overimpute . . . . .	9
set_python_env . . . . .	12
skip_if_no_numpy . . . . .	13
train . . . . .	13
undo_minmax . . . . .	15
<b>Index</b>	<b>17</b>

---

add_bin_labels	<i>Reverse numeric conversion of binary vector</i>
----------------	--

---

**Description**

Helper function to re-apply binary variable labels post-imputation.

**Usage**

```
add_bin_labels(x, one, zero)
```

**Arguments**

- x                    A numeric vector or column, scaled between 0 and 1
- one                  A character string, the label associated with binary value 1
- zero                A character string, the label associated with binary value 0

**Value**

Vector of character strings corresponding to binary values

**Examples**

```
ex_bin <- c(1,0,0,1,1,0,0,1,0)
cat <- "cat"
dog <- "dog"

add_bin_labels(x = ex_bin, one = cat, zero = dog)
```

---

coalesce_one_hot	<i>Coalesce one-hot encoding back to a single variable</i>
------------------	--

---

**Description**

Helper function to reverse one-hot encoding post-imputation.

**Usage**

```
coalesce_one_hot(X, var_name)
```

**Arguments**

X	A data.frame, data.table or matrix, for a single variable
var_name	A character string, with the original variable label

**Value**

A vector of length equal to nrow(X), containing categorical labels corresponding to the columns of X

---

col_minmax	<i>Scale numeric vector between 0 and 1</i>
------------	---

---

**Description**

Helper function to scale numeric variables. Aids convergence of Midas model.

**Usage**

```
col_minmax(x)
```

**Arguments**

x	A numeric vector or column.
---	-----------------------------

**Value**

Vector scaled between 0 and 1

**Examples**

```
ex_num <- runif(100,1,10)
scaled <- col_minmax(ex_num)
```

---

combine	<i>Estimate and combine regression models from multiply-imputed data</i>
---------	--

---

### Description

combine() calculates  $m$  individual regression models, then applies "Rubin's Rules" to produce a single, combined estimate of the regression parameters and uncertainty.

### Usage

```
combine(formula, df_list, dof_adjust = TRUE, ...)
```

### Arguments

formula	A formula, or character string coercible to a formula
df_list	A list, containing data.frames or objects coercible to data.frames
dof_adjust	Boolean, indicating whether or not to apply the Rubin and Barnard (1999) degrees of freedom adjustment for small-samples
...	Further arguments passed onto glm()

### Value

Data.frame of combined model results.

### Examples

```
set.seed(89)
test_dfs <- lapply(1:5, function (x) data.frame(a = rnorm(1000),
                                                b = runif(1000),
                                                c = 2*rnorm(1000)))

midas_res <- combine("a ~ b + c", df_list = test_dfs)
```

---

complete	<i>Impute missing values using imputation model</i>
----------	---

---

### Description

Having trained an imputation model, complete() produces  $m$  completed datasets, saved as a list.

**Usage**

```
complete(
  mid_obj,
  m = 10L,
  unscale = TRUE,
  bin_label = TRUE,
  cat_coalesce = TRUE,
  file = NULL,
  file_root = NULL
)
```

**Arguments**

<code>mid_obj</code>	Object of class <code>midas</code> , the result of running <code>rMIDAS::impute()</code>
<code>m</code>	An integer, the number of completed datasets required
<code>unscale</code>	Boolean, indicating whether to unscale any columns that were previously min-max scaled between 0 and 1
<code>bin_label</code>	Boolean, indicating whether to add back labels for binary columns
<code>cat_coalesce</code>	Boolean, indicating whether to decode the one-hot encoded categorical variables
<code>file</code>	Path to save completed datasets. If <code>NULL</code> , completed datasets are only loaded into memory.
<code>file_root</code>	A character string, used as the root for all filenames when saving completed datasets if a filepath is supplied. If no <code>file_root</code> is provided, saved datasets will be saved as <code>"file/midas_impute_yymmdd_hhmmss_m.csv"</code>

**Value**

List of length `m`, each element of which is a completed `data.frame` (i.e. no missing values)

**Examples**

```
# Generate raw data, with numeric, binary, and categorical variables

# Run where Python available
if (reticulate::py_module_available("numpy")) {

  raw_data <- data.table(a = sample(c("red", "yellow", "blue", NA), 1000, replace = TRUE),
                        b = 1:1000,
                        c = sample(c("YES", "NO", NA), 1000, replace = TRUE),
                        d = runif(1000, 1, 10),
                        e = sample(c("YES", "NO"), 1000, replace = TRUE),
                        f = sample(c("male", "female", "trans", "other", NA), 1000, replace = TRUE))

  # Names of bin./cat. variables
  test_bin <- c("c", "e")
  test_cat <- c("a", "f")

  # Pre-process data
```

```

test_data <- convert(raw_data,
                     bin_cols = test_bin,
                     cat_cols = test_cat,
                     minmax_scale = TRUE)

# Run imputations
test_imp <- train(test_data)

# Generate datasets
complete_datasets <- complete(test_imp, m = 5)

# Use Rubin's rules to combine m regression models
midas_pool <- combine(formula = d~a+c+e+f,
                     complete_datasets)
}

```

---

convert

*Pre-process data for Midas imputation*


---

## Description

convert pre-processes datasets to enable user-friendly interface with the main impute function.

## Usage

```
convert(data, bin_cols, cat_cols, minmax_scale = FALSE)
```

## Arguments

data	Either an object of class <code>data.frame</code> , <code>data.table</code> , or a path to a regular, delimited file
bin_cols, cat_cols	A vector, column names corresponding to binary and categorical variables respectively
minmax_scale	Boolean, indicating whether to scale all numeric columns between 0 and 1, to improve model convergence

## Details

The function has two advantages over manual pre-processing:

1. Utilises `data.table` for fast read-in and processing of large datasets
2. Outputs an object that can be passed directly to `impute` without re-specifying column names etc.

**Value**

Returns custom S3 object of class 'midas\_preproc' containing:

- data – processed version of input data,
- bin\_list – vector of binary variable names
- cat\_lists – embedded list of one-hot encoded categorical variable names
- minmax\_params – list of min. and max. values for each numeric object scaled

List containing converted data, categorical and binary labels to be imported into the imputation model, and scaling parameters for post-imputation transformations.

**Examples**

```
data = data.frame(a = sample(c("red", "yellow", "blue", NA), 100, replace = TRUE),
  b = 1:100,
  c = sample(c("YES", "NO", NA), 100, replace = TRUE),
  d = runif(100),
  e = sample(c("YES", "NO"), 100, replace = TRUE),
  f = sample(c("male", "female", "trans", "other", NA), 100, replace = TRUE),
  stringsAsFactors = FALSE)

bin <- c("c", "e")
cat <- c("a", "f")

convert(data, bin_cols = bin, cat_cols = cat)
```

---

import\_midas

*Instantiate Midas class*


---

**Description**

Import Midas class into R environment, and instantiates passed parameters.

**Usage**

```
import_midas(...)
```

**Arguments**

... Arguments passed to the MIDAS class for instantiating network

**Value**

Object of class 'midas'

---

`mid_py_setup`*Configure python for MIDAS imputation*

---

**Description**

This function checks if the required Python dependencies are installed, and if not, checks with user before installing them. This function is called automatically within `set_python_env`, and should only need to be used when manually configuring python installs using reticulate.

**Usage**

```
mid_py_setup()
```

---

`na_to_nan`*Replace NA missing values with NaN*

---

**Description**

Helper function to convert NA values in a `data.frame` to NaN. This ensures the correct conversion of missing values when reticulate converts R objects to their Python equivalent. See the reticulate package documentation on type conversions for more information.

**Usage**

```
na_to_nan(df)
```

**Arguments**

`df` Data frame, or object coercible to one.

**Value**

Data frame with NA values replaced with NaN values.

**Examples**

```
na_to_nan(data.frame(a = c(1, NA, 0, 0, NA, NA)))
```



overimpute

*Perform overimputation diagnostic test***Description**

`overimpute()` spikes additional missingness into the input data and reports imputation accuracy at training intervals specified by the user. `overimpute()` works like `train()` – users must specify input data, binary and categorical columns (if data is not generated via `convert()`), model parameters for the neural network, and then overimputation parameters (see below for full details).

**Usage**

```
overimpute(
  data,
  binary_columns = NULL,
  softmax_columns = NULL,
  spikein = 0.3,
  training_epochs,
  report_ival = 35,
  plot_vars = FALSE,
  skip_plot = FALSE,
  spike_seed = NULL,
  layer_structure = c(256, 256, 256),
  learn_rate = 4e-04,
  input_drop = 0.8,
  seed = 123L,
  latent_space_size = 4,
  cont_adj = 1,
  binary_adj = 1,
  softmax_adj = 1,
  dropout_level = 0.5,
  vae_layer = FALSE,
  vae_alpha = 1,
  vae_sample_var = 1
)
```

**Arguments**

- |                              |   |
|------------------------------|---|
| <code>data</code>            | A <code>data.frame</code> (or coercible) object, or an object of class <code>midas_pre</code> created from <code>rMIDAS::convert()</code>   |
| <code>binary_columns</code>  | A vector of column names, containing binary variables. NOTE: if data is a <code>midas_pre</code> object, this argument will be overwritten.   |
| <code>softmax_columns</code> | A list of lists, each internal list corresponding to a single categorical variable and containing names of the one-hot encoded variable names. NOTE: if data is a <code>midas_pre</code> object, this argument will be overwritten. |

spikein	A numeric between 0 and 1; the proportion of observed values in the input dataset to be randomly removed.
training_epochs	An integer, specifying the number of overimputation training epochs.
report_ival	An integer, specifying the number of overimputation training epochs between calculations of loss. Shorter intervals provide a more granular view of model performance but slow down the overimputation process.
plot_vars	Boolean, specifies whether to plot the distribution of original versus overimputed values. This takes the form of a density plot for continuous variables and a barplot for categorical variables (showing proportions of each class).
skip_plot	Boolean, specifies whether to suppress the main graphical output. This may be desirable when users are conducting a series of overimputation exercises and are primarily interested in the console output. <b>Note</b> , when skip_plot = FALSE, users must manually close the resulting pyplot window before the code will terminate.
spike_seed, seed	An integer, to initialize the pseudo-random number generators. Separate seeds can be provided for the spiked-in missingness and imputation, otherwise spike_seed is set to seed (default = 123L).
layer_structure	A vector of integers, The number of nodes in each layer of the network (default = c(256, 256, 256), denoting a three-layer network with 256 nodes per layer). Larger networks can learn more complex data structures but require longer training and are more prone to overfitting.
learn_rate	A number, the learning rate $\gamma$ (default = 0.0001), which controls the size of the weight adjustment in each training epoch. In general, higher values reduce training time at the expense of less accurate results.
input_drop	A number between 0 and 1. The probability of corruption for input columns in training mini-batches (default = 0.8). Higher values increase training time but reduce the risk of overfitting. In our experience, values between 0.7 and 0.95 deliver the best performance.
latent_space_size	An integer, the number of normal dimensions used to parameterize the latent space.
cont_adj	A number, weights the importance of continuous variables in the loss function
binary_adj	A number, weights the importance of binary variables in the loss function
softmax_adj	A number, weights the importance of categorical variables in the loss function
dropout_level	A number between 0 and 1, determines the number of nodes dropped to "thin" the network
vae_layer	Boolean, specifies whether to include a variational autoencoder layer in the network
vae_alpha	A number, the strength of the prior imposed on the Kullback-Leibler divergence term in the variational autoencoder loss functions.
vae_sample_var	A number, the sampling variance of the normal distributions used to parameterize the latent space.

## Details

Accuracy is measured as the RMSE of imputed values versus actual values for continuous variables and classification error for categorical variables (i.e., the fraction of correctly predicted classes subtracted from 1). Both metrics are reported in two forms:

1. their summed value over all Monte Carlo samples from the estimated missing-data posterior – "Aggregated RMSE" and "Aggregated softmax error";
2. their aggregated value divided by the number of such samples – "Individual RMSE" and "Individual softmax error".

In the final model, we recommend selecting the number of training epochs that minimizes the average value of these metrics — weighted by the proportion (or substantive importance) of continuous and categorical variables — in the overimputation exercise. This “early stopping” rule reduces the risk of overtraining and thus, in effect, serves as an extra layer of regularization in the network.

## Value

Object of class `midas`, and outputs both overimputation loss values to the console and generates overimputation graphs.

## See Also

`train` for the main imputation function.

## Examples

```
# Run where Python available
if (reticulate::py_module_available("numpy")) {

raw_data <- data.table(a = sample(c("red", "yellow", "blue", NA), 1000, replace = TRUE),
                      b = 1:1000,
                      c = sample(c("YES", "NO", NA), 1000, replace = TRUE),
                      d = runif(1000, 1, 10),
                      e = sample(c("YES", "NO"), 1000, replace = TRUE),
                      f = sample(c("male", "female", "trans", "other", NA), 1000, replace = TRUE))

# Names of bin./cat. variables
test_bin <- c("c", "e")
test_cat <- c("a", "f")

# Pre-process data
test_data <- convert(raw_data,
                    bin_cols = test_bin,
                    cat_cols = test_cat,
                    minmax_scale = TRUE)

# Overimpute - without plots
test_imp <- overimpute(test_data,
                      spikein = 0.3,
                      plot_vars = FALSE,
```

```

        skip_plot = TRUE,
        training_epochs = 10,
        report_ival = 5)
}

```

---

set\_python\_env

*Manually configure python version*


---

## Description

This function allows users to set a custom python binary, virtualenv, or Conda environment.

## Usage

```
set_python_env(path, type = "auto", exact = FALSE, ...)
```

## Arguments

path	Character string, path to python binary if type == "auto", path to virtualenv if type == "virtualenv", or the name of a Conda environment if type=="condaenv"
type	Character string, one of 'auto' (for python binary), 'virtualenv', or 'condaenv'
exact	Boolean. If TRUE then only load exact match from path, otherwise allow reticulate to scan for other versions.
...	Further argument passed to reticulate::use_condaenv() for conda executable if type == "condaenv".

## Details

Users comfortable with reticulate may wish to configure Python manually using reticulate, and then call mid\_py\_setup() to check/install required Python dependencies for rMIDAS.

## Value

Boolean indicating whether setup passed without errors

---

skip_if_no_numpy	<i>Skip test where 'numpy' not available.</i>
------------------	---

---

**Description**

Check if Python's numpy is available, and skip test if not. This function is called within some tests to ensure server tests involving reticulate calls execute properly.

**Usage**

```
skip_if_no_numpy()
```

**Value**

NULL

---

train	<i>Train an imputation model using Midas</i>
-------	--

---

**Description**

Build and run a MIDAS neural network on the supplied missing data.

**Usage**

```
train(
  data,
  binary_columns = NULL,
  softmax_columns = NULL,
  training_epochs = 10L,
  layer_structure = c(256, 256, 256),
  learn_rate = 4e-04,
  input_drop = 0.8,
  seed = 123L,
  latent_space_size = 4,
  cont_adj = 1,
  binary_adj = 1,
  softmax_adj = 1,
  dropout_level = 0.5,
  vae_layer = FALSE,
  vae_alpha = 1,
  vae_sample_var = 1
)
```

**Arguments**

<code>data</code>	A <code>data.frame</code> (or coercible) object, or an object of class <code>midas_pre</code> created from <code>rMIDAS::convert()</code>
<code>binary_columns</code>	A vector of column names, containing binary variables. NOTE: if <code>data</code> is a <code>midas_pre</code> object, this argument will be overwritten.
<code>softmax_columns</code>	A list of lists, each internal list corresponding to a single categorical variable and containing names of the one-hot encoded variable names. NOTE: if <code>data</code> is a <code>midas_pre</code> object, this argument will be overwritten.
<code>training_epochs</code>	An integer, indicating the number of forward passes to conduct when running the model.
<code>layer_structure</code>	A vector of integers, The number of nodes in each layer of the network (default = <code>c(256, 256, 256)</code> ), denoting a three-layer network with 256 nodes per layer). Larger networks can learn more complex data structures but require longer training and are more prone to overfitting.
<code>learn_rate</code>	A number, the learning rate $\gamma$ (default = 0.0001), which controls the size of the weight adjustment in each training epoch. In general, higher values reduce training time at the expense of less accurate results.
<code>input_drop</code>	A number between 0 and 1. The probability of corruption for input columns in training mini-batches (default = 0.8). Higher values increase training time but reduce the risk of overfitting. In our experience, values between 0.7 and 0.95 deliver the best performance.
<code>seed</code>	An integer, the value to which Python's pseudo-random number generator is initialized. This enables users to ensure that data shuffling, weight and bias initialization, and missingness indicator vectors are reproducible.
<code>latent_space_size</code>	An integer, the number of normal dimensions used to parameterize the latent space.
<code>cont_adj</code>	A number, weights the importance of continuous variables in the loss function
<code>binary_adj</code>	A number, weights the importance of binary variables in the loss function
<code>softmax_adj</code>	A number, weights the importance of categorical variables in the loss function
<code>dropout_level</code>	A number between 0 and 1, determines the number of nodes dropped to "thin" the network
<code>vae_layer</code>	Boolean, specifies whether to include a variational autoencoder layer in the network
<code>vae_alpha</code>	A number, the strength of the prior imposed on the Kullback-Leibler divergence term in the variational autoencoder loss functions.
<code>vae_sample_var</code>	A number, the sampling variance of the normal distributions used to parameterize the latent space.

**Value**

Object of class `midas` from which completed datasets can be drawn, using `rMIDAS::complete()`

## Examples

```
# Generate raw data, with numeric, binary, and categorical variables

# Run where Python available
if (reticulate::py_module_available("numpy")) {

  raw_data <- data.table(a = sample(c("red", "yellow", "blue", NA), 1000, replace = TRUE),
                        b = 1:1000,
                        c = sample(c("YES", "NO", NA), 1000, replace = TRUE),
                        d = runif(1000, 1, 10),
                        e = sample(c("YES", "NO"), 1000, replace = TRUE),
                        f = sample(c("male", "female", "trans", "other", NA), 1000, replace = TRUE))

  # Names of bin./cat. variables
  test_bin <- c("c", "e")
  test_cat <- c("a", "f")

  # Pre-process data
  test_data <- convert(raw_data,
                      bin_cols = test_bin,
                      cat_cols = test_cat,
                      minmax_scale = TRUE)

  # Run imputations
  test_imp <- train(test_data)

  # Generate datasets
  complete_datasets <- complete(test_imp, m = 5)

  # Use Rubin's rules to combine m regression models
  midas_pool <- combine(formula = d~a+c+e+f,
                       complete_datasets)
}
```

---

undo\_minmax

*Reverse minmax scaling of numeric vector*


---

## Description

Helper function to reverse minmax scaling applied in the pre-processing step.

## Usage

```
undo_minmax(s, s_min, s_max)
```

**Arguments**

<code>s</code>	A numeric vector or column, scaled between 0 and 1.
<code>s_min</code>	A numeric value, the minimum of the unscaled vector
<code>s_max</code>	A numeric value, the maximum of the unscaled vector

**Value**

Vector re-scaled using original parameters `s_min` and `s_max`

**Examples**

```
ex_num <- runif(100,1,10)
scaled <- col_minmax(ex_num)
undo_scale <- undo_minmax(scaled, s_min = min(ex_num), s_max = max(ex_num))

# Prove two are identical
all.equal(ex_num, undo_scale)
```



# Index

- \* **diagnostics**
  - overimpute, [9](#)
- \* **import**
  - combine, [4](#)
- \* **imputation**
  - complete, [4](#)
  - import\_midas, [7](#)
  - train, [13](#)
- \* **postprocessing**
  - add\_bin\_labels, [2](#)
  - coalesce\_one\_hot, [3](#)
  - undo\_minmax, [15](#)
- \* **preprocessing**
  - col\_minmax, [3](#)
  - convert, [6](#)
  - na\_to\_nan, [8](#)
- \* **setup**
  - mid\_py\_setup, [8](#)
  - set\_python\_env, [12](#)
  - skip\_if\_no\_numpy, [13](#)

[add\\_bin\\_labels, 2](#)

[coalesce\\_one\\_hot, 3](#)

[col\\_minmax, 3](#)

[combine, 4](#)

[complete, 4](#)

[convert, 6](#)

[import\\_midas, 7](#)

[mid\\_py\\_setup, 8](#)

[na\\_to\\_nan, 8](#)

[overimpute, 9](#)

[set\\_python\\_env, 12](#)

[skip\\_if\\_no\\_numpy, 13](#)

[train, 11, 13](#)

[undo\\_minmax, 15](#)