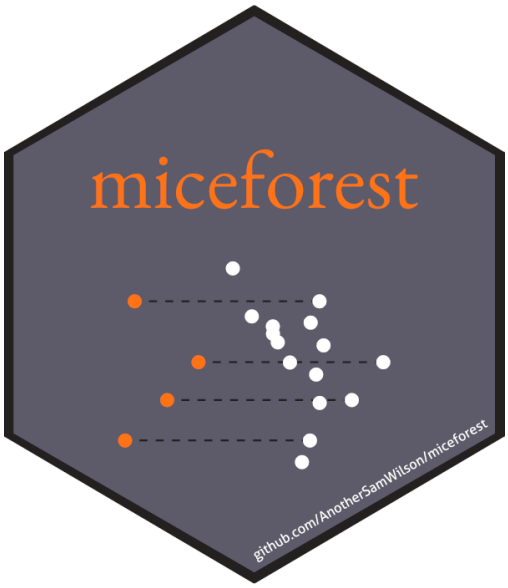
miceforest

Release 2021-08-21

Samuel Von Wilson

Dec 12, 2022

CONTENTS:



`miceforest` imputes missing data using LightGBM in an iterative method known as Multiple Imputation by Chained Equations (MICE). It was designed to be:

- **Fast**
  - Uses lightgbm as a backend
  - Has efficient mean matching solutions.
  - Can utilize GPU training
- **Flexible**
  - Can impute pandas dataframes and numpy arrays
  - Handles categorical data automatically
  - Fits into a sklearn pipeline
  - User can customize every aspect of the imputation process
- **Production Ready**
  - Can impute new, unseen datasets very quickly
  - Kernels are efficiently compressed during saving and loading
  - Data can be imputed in place to save memory
  - Can build models on non-missing data

There are very extensive beginner and advanced tutorials on the github readme. Below is a table of contents for the topics covered:

CHAPTER

ONE

IMPUTATIONKERNEL

1.1 ImputationKernel

ImputationKernel(data[, datasets, ...])	Creates a kernel dataset.

1.1.1 miceforest.ImputationKernel

class miceforest.**ImputationKernel**(*data*, *datasets=1*, *variable_schema=None*, *imputation_order='ascending'*, *train_nonmissing=False*, *mean_match_scheme=None*, *data_subset=None*, *categorical_feature='auto'*, *initialization='random'*, *save_all_iterations=True*, *save_models=1*, *copy_data=True*, *save_loggers=False*, *random_state=None*)

Bases: *miceforest.ImputedData.ImputedData*

Creates a kernel dataset. This dataset can perform MICE on itself, and impute new data from models obtained during MICE.

Parameters

- **data** (*np.ndarray or pandas DataFrame.*) –

```
The data to be imputed.
```

- **variable_schema** (*None or list or dict, default=None*) –

```
Specifies the feature - target relationships used to train models.
This parameter also controls which models are built. Models can be␣
↪built
even if a variable contains no missing values, or is not being␣
↪imputed
(train_nonmissing must be set to True).

    - If None, all columns will be used as features in the training␣
↪of each model.
    - If list, all columns in data are used to impute the variables␣
↪in the list
    - If dict the values will be used to impute the keys. Can be␣
↪either column
        indices or names (if data is a pd.DataFrame).
```

(continues on next page)

(continued from previous page)

```
No models will be trained for variables not specified by variable_
↪schema
(either by None, a list, or in dict keys).
```

- **imputation_order** (*str, list[str], list[int], default="ascending"*) –

```
The order the imputations should occur in. If a string from the
items below, all variables specified by variable_schema with
missing data are imputed:
    ascending: variables are imputed from least to most missing
    descending: most to least missing
    roman: from left to right in the dataset
    arabic: from right to left in the dataset.
If a list is provided:
    - the variables will be imputed in that order.
    - only variables with missing values should be included in the␣
↪list.
    - must be a subset of variables specified by variable_schema.
If a variable with missing values is in variable_schema, but not in
imputation_order, then models to impute that variable will be␣
↪trained,
but the actual values will not be imputed. See examples for details.
```

- **train_nonmissing** (*boolean*) –

```
Should models be trained for variables with no missing values?␣
↪Useful if you
expect you will need to impute new data which will have missing␣
↪values, but
the training data is fully recognized.

If True, parameters are interpreted like so:
    - models are run for all variables specified by variable_schema
    - if variable_schema is None, models are run for all variables
    - each iteration, models build for fully recognized variables are
        always trained after the models trained during mice.
    - imputation_order does not have any affect on fully recognized
        variable model training.

WARNING: Setting this to True without specifying a variable schema␣
↪will build
models for all variables in the dataset, whether they have missing␣
↪values or
not. This may or may not be what you want.
```

- **data_subset** (*None or int or float or dict.*) –

```
Subsets the data used in each iteration, which can save a␣
↪significant amount of time.
This can also help with memory consumption, as the candidate data␣
↪must be copied to
make a feature dataset for lightgbm.
```

(continues on next page)

(continued from previous page)

```
The number of rows used for each variable is (# rows in raw data) - (
↪# missing variable values)
for each variable. data_subset takes a random sample of this.

If float, must be 0.0 < data_subset <= 1.0. Interpreted as a␣
↪percentage of available candidates
If int must be data_subset >= 0. Interpreted as the number of␣
↪candidates.
If 0, no subsetting is done.
If dict, keys must be variable names, and values must follow two␣
↪above rules.

It is recommended to carefully select this value for each variable␣
↪if dealing
with very large data that barely fits into memory.
```

- **mean_match_scheme** (*Dict, default = None*) –

```
An instance of the miceforest.MeanMatchScheme class.

If None is passed, a sensible default scheme is used. There are␣
↪multiple helpful
schemes that can be accessed from miceforest.builtin_mean_match_
↪schemes, or
you can build your own.

A description of the defaults:
- mean_match_default (default, if mean_match_scheme is None))
    This scheme is has medium speed and accuracy for most data.

    Categorical:
        If mmc = 0, the class with the highest probability is chosen.
        If mmc > 0, get N nearest neighbors from class probabilities.
            Select 1 at random.
    Numeric:
        If mmc = 0, the predicted value is used
        If mmc > 0, obtain the mmc closest candidate
            predictions and collect the associated
            real candidate values. Choose 1 randomly.

- mean_match_shap
    This scheme is the most accurate, but takes the longest.
    It works the same as mean_match_default, except all nearest
    neighbor searches are performed on the shap values of the
    predictions, instead of the predictions themselves.

- mean_match_scheme_fast_cat:
    This scheme is faster for categorical variables,
    but may be less accurate as well..

    Categorical:
```

(continues on next page)

(continued from previous page)

```
        If mmc = 0, the class with the highest probability is chosen.
        If mmc > 0, return class based on random draw weighted by
            class probability for each sample.
    Numeric or binary:
        If mmc = 0, the predicted value is used
        If mmc > 0, obtain the mmc closest candidate
            predictions and collect the associated
            real candidate values. Choose 1 randomly.
```

- **categorical_feature** (*str or list, default="auto"*) –

```
The categorical features in the dataset. This handling depends on␣
↪class of impute_data:

    pandas DataFrame:
        - "auto": categorical information is inferred from any␣
↪columns with
            datatype category or object.
        - list of column names (or indices): Useful if all␣
↪categorical columns
            have already been cast to numeric encodings of some type,
↪ otherwise you
            should just use "auto". Will throw an error if a list is␣
↪provided AND
            categorical dtypes exist in data. If a list is provided,␣
↪values in the
            columns must be consecutive integers starting at 0, as␣
↪required by lightgbm.

    numpy ndarray:
        - "auto": no categorical information is stored.
        - list of column indices: Specified columns are treated as␣
↪categorical. Column
            values must be consecutive integers starting at 0, as␣
↪required by lightgbm.
```

- **initialization** (*str*) –

```
"random" - missing values will be filled in randomly from existing␣
↪values.
"empty" - lightgbm will start MICE without initial imputation
```

- **save_all_iterations** (*boolean, optional(default=True)*) –

```
Save all the imputation values from all iterations, or just
the latest. Saving all iterations allows for additional
plotting, but may take more memory
```

- **save_models** (*int*) –

```
Which models should be saved:
    = 0: no models are saved. Cannot get feature importance or
        impute new data.
```

(continues on next page)

(continued from previous page)

```
    = 1: only the last model iteration is saved. Can only get
        feature importance of last iteration. New data is
        imputed using the last model for all specified iterations.
        This is only an issue if data is heavily Missing At Random.
    = 2: all model iterations are saved. Can get feature importance
        for any iteration. When imputing new data, each iteration is
        imputed using the model obtained at that iteration in mice.
        This allows for imputations that most closely resemble those
        that would have been obtained in mice.
```

- **copy_data** (*boolean (default = False)*) –

```
Should the dataset be referenced directly? If False, this will cause
the dataset to be altered in place. If a copy is created, it is saved
in self.working_data. There are different ways in which the dataset
can be altered:

1) complete_data() will fill in missing values
2) To save space, mice() references and manipulates self.working_
↪data directly.
    If self.working_data is a reference to the original dataset, the␣
↪original
    dataset will undergo these manipulations during the mice process.
    At the end of the mice process, missing values will be set back␣
↪to np.NaN
    where they were originally missing.
```

- **save_loggers** (*boolean (default = False)*) –

```
A logger is created each time mice() or impute_new_data() is called.
If True, the loggers are stored in a list ImputationKernel.loggers.
If you wish to start saving logs, call ImputationKernel.start_
↪logging().
If you wish to stop saving logs, call ImputationKernel.stop_
↪logging().
```

- **random_state** (*None,int, or numpy.random.RandomState*) –

```
The random_state ensures script reproducibility. It only ensures␣
↪reproducible
results if the same script is called multiple times. It does not␣
↪guarantee
reproducible results at the record level, if a record is imputed␣
↪multiple
different times. If reproducible record-results are desired, a seed␣
↪must be
passed for each record in the random_seed_array parameter.
```

__init__(*data*, *datasets=1*, *variable_schema=None*, *imputation_order='ascending'*, *train_nonmissing=False*, *mean_match_scheme=None*, *data_subset=None*, *categorical_feature='auto'*, *initialization='random'*, *save_all_iterations=True*, *save_models=1*, *copy_data=True*, *save_loggers=False*, *random_state=None*)

Methods

`__init__`(data[, datasets, variable_schema, ...])	

`append`(imputation_kernel)	Combine two imputation kernels together.
`compile_candidate_preds`()	Candidate predictions can be pre-generated before imputing new data.
`complete_data`([dataset, iteration, inplace, ...])	Return dataset with missing values imputed.
`dataset_count`()	Return the number of datasets.
`delete_candidate_preds`()	Deletes the pre-computed candidate predictions.
`fit`(X, y, **fit_params)	Method for fitting a kernel when used in a sklearn pipeline.
`get_correlations`(datasets, variables)	Return the correlations between datasets for the specified variables.
`get_feature_importance`(dataset[, iteration])	Return a matrix of feature importance.
`get_means`(datasets[, variables])	Return a dict containing the average imputation value for specified variables at each iteration.
`get_model`(dataset, variable[, iteration])	Return the model for a specific dataset, variable, iteration.
`get_raw_prediction`(variable[, imp_dataset, ...])	Get the raw model output for a specific variable.
`impute_new_data`(new_data[, datasets, ...])	Impute a new dataset
`iteration_count`([datasets, variables])	Grabs the iteration count for specified variables, datasets.
`mice`([iterations, verbose, ...])	Perform mice on a given dataset.
`plot_correlations`([datasets, variables])	Plot the correlations between datasets.
`plot_feature_importance`(dataset[, ...])	Plot the feature importance.
`plot_imputed_distributions`([datasets, ...])	Plot the imputed value distributions.
`plot_mean_convergence`([datasets, variables])	Plots the average value of imputations over each iteration.
`save_kernel`(filepath[, clevel, cname, ...])	Compresses and saves the kernel to a file.
`start_logging`()	Start saving loggers to self.loggers
`stop_logging`()	Stop saving loggers to self.loggers
`transform`(X[, y])	Method for calling a kernel when used in a sklearn pipeline.
`tune_parameters`(dataset[, variables, ...])	Perform hyperparameter tuning on models at the current iteration.

append(*imputation_kernel*)

Combine two imputation kernels together. For compatibility, the following attributes of each must be equal:

- working_data
- iteration_count
- categorical_feature
- mean_match_scheme
- variable_schema
- imputation_order
- save_models
- save_all_iterations

Only cursory checks are done to ensure working_data is equal. Appending a kernel with different working_data could ruin this kernel.

Parameters **imputation_kernel** (ImputationKernel) – The kernel to merge.

compile_candidate_preds()
Candidate predictions can be pre-generated before imputing new data. This can save a substantial amount of time, especially if save_models == 1.

complete_data(*dataset=0*, *iteration=None*, *inplace=False*, *variables=None*)
Return dataset with missing values imputed.

Parameters

- **dataset** (*int*) – The dataset to complete.
- **iteration** (*int*) – Impute data with values obtained at this iteration. If None, returns the most up-to-date iterations, even if different between variables. If not none, iteration must have been saved in imputed values.
- **inplace** (*bool*) – Should the data be completed in place? If True, self.working_data is imputed,and nothing is returned. This is useful if the dataset is very large. If False, a copy of the data is returned, with missing values imputed.

Returns

Return type The completed data, with values imputed for specified variables.

dataset_count()
Return the number of datasets. Datasets are defined by how many different sets of imputation values we have accumulated.

delete_candidate_preds()
Deletes the pre-computed candidate predictions.

fit(*X*, *y*, ***fit_params*)
Method for fitting a kernel when used in a sklearn pipeline. Should not be called by the user directly.

get_correlations(*datasets*, *variables*)
Return the correlations between datasets for the specified variables.

Parameters **variables** (*list[str], list[int]*) – The variables to return the correlations for.

Returns The correlations at each iteration for the specified variables.

Return type dict

get_feature_importance(*dataset*, *iteration=None*)
Return a matrix of feature importance. The cells represent the normalized feature importance of the columns to impute the rows. This is calculated internally by lightgbm.Booster.feature_importance().

Parameters

- **dataset** (*int*) – The dataset to get the feature importance for.
- **iteration** (*int*) – The iteration to return the feature importance for. Right now, the model must be saved to return importance

Returns

- *np.ndarray of importance values. Rows are imputed variables, and*
- *columns are predictor variables.*

get_means(*datasets*, *variables=None*)
Return a dict containing the average imputation value for specified variables at each iteration.

get_model(*dataset*, *variable*, *iteration=None*)
Return the model for a specific dataset, variable, iteration.

Parameters

- **dataset** (*int*) – The dataset to return the model for.
- **var** (*str*) – The variable that was imputed
- **iteration** (*int*) – The model iteration to return. Keep in mind if save_models ==1, the model was not saved. If none is provided, the latest model is returned.
- **Returns** (*lightgbm.Booster*) – The model used to impute this specific variable, iteration.

get_raw_prediction(*variable*, *imp_dataset=0*, *imp_iteration=None*, *model_dataset=None*, *model_iteration=None*, *dtype=None*)
Get the raw model output for a specific variable.

The data is pulled from the imp_dataset dataset, at the imp_iteration iteration. The model is pulled from model_dataset dataset, at the model_iteration iteration.

So, for example, it is possible to get predictions using the imputed values for dataset 3, at iteration 2, using the model obtained from dataset 10, at iteration 6. This is assuming desired iterations and models have been saved.

Parameters

- **variable** (*int or str*) – The variable to get the raw predictions for. Can be an index or variable name.
- **imp_dataset** (*int*) – The imputation dataset to use when creating the feature dataset.
- **imp_iteration** (*int*) – The iteration from which to draw the imputation values when creating the feature dataset. If None, the latest iteration is used.
- **model_dataset** (*int*) – The dataset from which to pull the trained model for this variable. If None, it is selected to be the same as imp_dataset.
- **model_iteration** (*int*) – The iteration from which to pull the trained model for this variable If None, it is selected to be the same as imp_iteration.
- **dtype** (*str, np.dtype*) – The datatype to cast the raw prediction as. Passed to MeanMatchScheme.model_predict().

Returns

Return type np.ndarray of raw predictions.

impute_new_data(*new_data*, *datasets=None*, *iterations=None*, *save_all_iterations=True*, *copy_data=True*, *random_state=None*, *random_seed_array=None*, *verbose=False*)
Impute a new dataset

Uses the models obtained while running MICE to impute new data, without fitting new models. Pulls mean matching candidates from the original data.

save_models must be > 0. If save_models == 1, the last model obtained in mice is used for every iteration. If save_models > 1, the model obtained at each iteration is used to impute the new data for that iteration. If specified iterations is greater than the number of iterations run so far using mice, the last model is used for each additional iteration.

Type checking is not done. It is up to the user to ensure that the kernel data matches the new data being imputed.

Parameters

- **new_data** (*pandas DataFrame or numpy ndarray*) – The new data to impute
- **datasets** (*int or List[int] (default = None)*) – The datasets from the kernel to use to impute the new data. If None, all datasets from the kernel are used.
- **iterations** (*int*) – The number of iterations to run. If None, the same number of iterations run so far in mice is used.
- **save_all_iterations** (*bool*) – Should the imputation values of all iterations be archived? If False, only the latest imputation values are saved.
- **copy_data** (*boolean*) – Should the dataset be referenced directly? This will cause the dataset to be altered in place. If a copy is created, it is saved in self.working_data. There are different ways in which the dataset can be altered:

  1) complete_data() will fill in missing values

  2) mice() references and manipulates self.working_data directly.

- **random_state** (*int or np.random.RandomState or None (default=None)*) – The random state of the process. Ensures reproducibility. If None, the random state of the kernel is used. Beware, this permanently alters the random state of the kernel and ensures non-reproduceable results, unless the entire process up to this point is re-run.
- **random_seed_array** (*None or np.ndarray (int32)*) –

```
Record-level seeds.

Ensures deterministic imputations at the record level. random_seed_
↪array causes
deterministic imputations for each record no matter what dataset␣
↪each record is
imputed with, assuming the same number of iterations and datasets␣
↪are used.
If random_seed_array os passed, random_state must also be passed.

Record-level imputations are deterministic if the following␣
↪conditions are met:
    1) The associated seed is the same.
    2) The same kernel is used.
    3) The same number of iterations are run.
    4) The same number of datasets are run.

Notes:
    a) This will slightly slow down the imputation process, because␣
↪random
    number generation in numpy can no longer be vectorized. If you␣
↪don't have a
    specific need for deterministic imputations at the record level,
↪ it is better to
    keep this parameter as None.

    b) Using this parameter may change the global numpy seed by␣
↪calling np.random.seed().
```

(continues on next page)

(continued from previous page)

```
   c) Internally, these seeds are hashed each time they are used,␣
↪in order
   to obtain different results for each dataset / iteration.
```

verbose: boolean Should information about the process be printed?

Returns

Return type *miceforest.ImputedData*

`iteration_count`(*datasets=None*, *variables=None*)

Grabs the iteration count for specified variables, datasets. If the iteration count is not consistent across the provided datasets/variables, an error will be thrown. Providing None will use all datasets/variables.

This is to ensure the process is in a consistent state when the iteration count is needed.

Parameters

- **datasets** (*int or list[int]*) – The datasets to check the iteration count for.
- **variables** (*int, str, list[int] or list[str]:*) – The variables to check the iteration count for. Variables can be specified by their names or indexes.

Returns

Return type An integer representing the iteration count.

`mice`(*iterations=2*, *verbose=False*, *variable_parameters=None*, *compile_candidates=False*, ***kwlgb*)

Perform mice on a given dataset.

Multiple Imputation by Chained Equations (MICE) is an iterative method which fills in (imputes) missing data points in a dataset by modeling each column using the other columns, and then inferring the missing data.

For more information on MICE, and missing data in general, see Stef van Buuren's excellent online book: https://stefvanbuuren.name/fimd/ch-introduction.html

For detailed usage information, see this project's README on the github repository: https://github.com/AnotherSamWilson/miceforest

Parameters

- **iterations** (*int*) – The number of iterations to run.
- **verbose** (*bool*) – Should information about the process be printed?
- **variable_parameters** (*None or dict*) – Model parameters can be specified by variable here. Keys should be variable names or indices, and values should be a dict of parameter which should apply to that variable only.
- **compile_candidates** (*bool*) – Candidate predictions can be stored as they are created while performing mice. This prevents kernel.compile_candidate_preds() from having to be called separately, and can save a significant amount of time if compiled candidate predictions are desired.
- **kwlgb** – Additional arguments to pass to lightgbm. Applied to all models.

`plot_correlations`(*datasets=None*, *variables=None*, ***adj_args*)

Plot the correlations between datasets. See get_correlations() for more details.

Parameters

- **datasets** (*None or list[int]*) – The datasets to plot.
- **variables** (*None,list*) – The variables to plot.
- **adj_args** – Additional arguments passed to plt.subplots_adjust()

plot_feature_importance(*dataset*, *normalize=True*, *iteration=None*, ***kw_plot*)
Plot the feature importance. See get_feature_importance() for more details.

Parameters

- **dataset** (*int*) – The dataset to plot the feature importance for.
- **iteration** (*int*) – The iteration to plot the feature importance of.
- **normalize** (*book*) – Should the values be normalize from 0-1? If False, values are raw from Booster.feature_importance()
- **kw_plot** – Additional arguments sent to sns.heatmap()

plot_imputed_distributions(*datasets=None*, *variables=None*, *iteration=None*, ***adj_args*)
Plot the imputed value distributions. Red lines are the distribution of original data Black lines are the distribution of the imputed values.

Parameters

- **datasets** (*None, int, list[int]*) –
- **variables** (*None, str, int, list[str], or list[int]*) – The variables to plot. If None, all numeric variables are plotted.
- **iteration** (*None, int*) – The iteration to plot the distribution for. If None, the latest iteration is plotted. save_all_iterations must be True if specifying an iteration.
- **adj_args** – Additional arguments passed to plt.subplots_adjust()

plot_mean_convergence(*datasets=None*, *variables=None*, ***adj_args*)
Plots the average value of imputations over each iteration.

Parameters

- **variables** (*None or list*) – The variables to plot. Must be numeric.
- **adj_args** – Passed to matplotlib.pyplot.subplots_adjust()

save_kernel(*filepath*, *clevel=None*, *cname=None*, *n_threads=None*, *copy_while_saving=True*)
Compresses and saves the kernel to a file.

Parameters

- **filepath** (*str*) – The file to save to.
- **clevel** (*int*) – The compression level, sent to clevel argument in blosc.compress()
- **cname** (*str*) – The compression algorithm used. Sent to cname argument in blosc.compress. If None is specified, the default is lz4hc.
- **n_threads** (*int*) – The number of threads to use for compression. By default, all threads are used.
- **copy_while_saving** (*boolean*) – Should the kernel be copied while saving? Copying is safer, but may take more memory.

start_logging()
Start saving loggers to self.loggers

stop_logging()
Stop saving loggers to self.loggers

transform(*X*, *y=None*)
Method for calling a kernel when used in a sklearn pipeline. Should not be called by the user directly.

tune_parameters(*dataset*, *variables=None*, *variable_parameters=None*, *parameter_sampling_method='random'*, *nfold=10*, *optimization_steps=5*, *random_state=None*, *verbose=False*, ***kwbounds*)
Perform hyperparameter tuning on models at the current iteration.

```
A few notes:
- Underlying models will now be gradient boosted trees by default (or any
    other boosting type compatible with lightgbm.cv).
- The parameters are tuned on the data that would currently be returned by
    complete_data(dataset). It is usually a good idea to run at least 1␣
↪iteration
    of mice with the default parameters to get a more accurate idea of the
    real optimal parameters, since Missing At Random (MAR) data imputations
    tend to converge over time.
- num_iterations is treated as the maximum number of boosting rounds to run
    in lightgbm.cv. It is NEVER optimized. The num_iterations that is returned
    is the best_iteration returned by lightgbm.cv. num_iterations can be passed␣
↪to
    limit the boosting rounds, but the returned value will always be obtained
    from best_iteration.
- lightgbm parameters are chosen in the following order of priority:
    1) Anything specified in variable_parameters
    2) Parameters specified globally in **kwbounds
    3) Default tuning space (miceforest.default_lightgbm_parameters.make_
↪default_tuning_space)
    4) Default parameters (miceforest.default_lightgbm_parameters.default_
↪parameters)
- See examples for a detailed run-through. See
    https://github.com/AnotherSamWilson/miceforest#Tuning-Parameters
    for even more detailed examples.
```

Parameters

- **dataset** (*int (required)*) –

```
The dataset to run parameter tuning on. Tuning parameters on 1␣
↪dataset usually results
in acceptable parameters for all datasets. However, tuning results␣
↪are still stored
seperately for each dataset.
```

- **variables** (*None or list*) –

```
- If None, default hyper-parameter spaces are selected based on␣
↪kernel data, and
all variables with missing values are tuned.
- If list, must either be indexes or variable names corresponding␣
↪to the variables
that are to be tuned.
```

- **variable_parameters** (*None or dict*) –

```
Defines the tuning space. Dict keys must be variable names or␣
↪indices, and a subset
of the variables parameter. Values must be a dict with lightgbm␣
↪parameter names as
keys, and values that abide by the following rules:
    scalar: If a single value is passed, that parameter will be␣
↪used to build the
        model, and will not be tuned.
    tuple: If a tuple is passed, it must have length = 2 and will␣
↪be interpreted as
        the bounds to search within for that parameter.
    list: If a list is passed, values will be randomly selected␣
↪from the list.
        NOTE: This is only possible with method = 'random'.

example: If you wish to tune the imputation model for the 4th␣
↪variable with specific
bounds and parameters, you could pass:
    variable_parameters = {
        4: {
            'learning_rate: 0.01',
            'min_sum_hessian_in_leaf: (0.1, 10),
            'extra_trees': [True, False]
        }
    }
All models for variable 4 will have a learning_rate = 0.01. The␣
↪process will randomly
search within the bounds (0.1, 10) for min_sum_hessian_in_leaf, and␣
↪extra_trees will
be randomly selected from the list. Also note, the variable name␣
↪for the 4th column
could also be passed instead of the integer 4. All other variables␣
↪will be tuned with
the default search space, unless **kwbounds are passed.
```

- **parameter_sampling_method** (*str*) –

```
If 'random', parameters are randomly selected.
Other methods will be added in future releases.
```

- **nfold** (*int*) –

```
The number of folds to perform cross validation with. More folds␣
↪takes longer, but
Gives a more accurate distribution of the error metric.
```

- **optimization_steps** –

```
How many steps to run the process for.
```

- **random_state** (*int or np.random.RandomState or None (default=None)*) –

```
The random state of the process. Ensures reproduceability. If None,␣
↪the random state
of the kernel is used. Beware, this permanently alters the random␣
↪state of the kernel
and ensures non-reproduceable results, unless the entire process up␣
↪to this point
is re-run.
```

- **kwbounds** –

```
Any additional arguments that you want to apply globally to every␣
↪variable.
For example, if you want to limit the number of iterations, you␣
↪could pass
num_iterations = x to this functions, and it would apply globally.␣
↪Custom
bounds can also be passed.
```

Returns

- **2 dicts** (*optimal_parameters, optimal_parameter_losses*)
- **- optimal_parameters** (*dict*) – A dict of the optimal parameters found for each variable. This can be passed directly to the variable_parameters parameter in mice()

```
{variable: {parameter_name: parameter_value}}
```

- **- optimal_parameter_losses** (*dict*) – The average out of fold cv loss obtained directly from lightgbm.cv() associated with the optimal parameter set.

```
{variable: loss}
```

CHAPTER

TWO

IMPUTEDDATA

2.1 ImputedData

ImputedData(impute_data[, datasets, ...])	Imputed Data

2.1.1 miceforest.ImputedData

class miceforest.**ImputedData**(*impute_data*, *datasets=5*, *variable_schema=None*, *imputation_order='ascending'*, *train_nonmissing=False*, *categorical_feature='auto'*, *save_all_iterations=True*, *copy_data=True*)

Bases: `object`

Imputed Data

This class should not be instantiated directly. Instead, it is returned when ImputationKernel.impute_new_data() is called. For parameter arguments, see ImputationKernel documentation.

__init__(*impute_data*, *datasets=5*, *variable_schema=None*, *imputation_order='ascending'*, *train_nonmissing=False*, *categorical_feature='auto'*, *save_all_iterations=True*, *copy_data=True*)

Methods

`__init__`(impute_data[, datasets, ...])	

`complete_data`([dataset, iteration, inplace, ...])	Return dataset with missing values imputed.
`dataset_count`()	Return the number of datasets.
`get_correlations`(datasets, variables)	Return the correlations between datasets for the specified variables.
`get_means`(datasets[, variables])	Return a dict containing the average imputation value for specified variables at each iteration.
`iteration_count`([datasets, variables])	Grabs the iteration count for specified variables, datasets.
`plot_correlations`([datasets, variables])	Plot the correlations between datasets.
`plot_imputed_distributions`([datasets, ...])	Plot the imputed value distributions.
`plot_mean_convergence`([datasets, variables])	Plots the average value of imputations over each iteration.

complete_data(*dataset=0*, *iteration=None*, *inplace=False*, *variables=None*)

Return dataset with missing values imputed.

Parameters

- **dataset** (*int*) – The dataset to complete.
- **iteration** (*int*) – Impute data with values obtained at this iteration. If None, returns the most up-to-date iterations, even if different between variables. If not none, iteration must have been saved in imputed values.
- **inplace** (*bool*) – Should the data be completed in place? If True, self.working_data is imputed,and nothing is returned. This is useful if the dataset is very large. If False, a copy of the data is returned, with missing values imputed.

Returns

Return type The completed data, with values imputed for specified variables.

dataset_count()

Return the number of datasets. Datasets are defined by how many different sets of imputation values we have accumulated.

get_correlations(*datasets*, *variables*)

Return the correlations between datasets for the specified variables.

Parameters **variables** (*list[str], list[int]*) – The variables to return the correlations for.

Returns The correlations at each iteration for the specified variables.

Return type dict

get_means(*datasets*, *variables=None*)

Return a dict containing the average imputation value for specified variables at each iteration.

iteration_count(*datasets=None*, *variables=None*)

Grabs the iteration count for specified variables, datasets. If the iteration count is not consistent across the provided datasets/variables, an error will be thrown. Providing None will use all datasets/variables.

This is to ensure the process is in a consistent state when the iteration count is needed.

Parameters

- **datasets** (*int or list[int]*) – The datasets to check the iteration count for.
- **variables** (*int, str, list[int] or list[str]:*) – The variables to check the iteration count for. Variables can be specified by their names or indexes.

Returns

Return type An integer representing the iteration count.

plot_correlations(*datasets=None*, *variables=None*, ***adj_args*)

Plot the correlations between datasets. See get_correlations() for more details.

Parameters

- **datasets** (*None or list[int]*) – The datasets to plot.
- **variables** (*None,list*) – The variables to plot.
- **adj_args** – Additional arguments passed to plt.subplots_adjust()

plot_imputed_distributions(*datasets=None*, *variables=None*, *iteration=None*, ***adj_args*)

Plot the imputed value distributions. Red lines are the distribution of original data Black lines are the distribution of the imputed values.

Parameters

- **datasets** (*None, int, list[int]*) –
- **variables** (*None, str, int, list[str], or list[int]*) – The variables to plot. If None, all numeric variables are plotted.
- **iteration** (*None, int*) – The iteration to plot the distribution for. If None, the latest iteration is plotted. save_all_iterations must be True if specifying an iteration.
- **adj_args** – Additional arguments passed to plt.subplots_adjust()

plot_mean_convergence(*datasets=None*, *variables=None*, ***adj_args*)

Plots the average value of imputations over each iteration.

Parameters

- **variables** (*None or list*) – The variables to plot. Must be numeric.
- **adj_args** – Passed to matplotlib.pyplot.subplots_adjust()

CHAPTER

THREE

MEANMATCHSCHEME

3.1 MeanMatchScheme

MeanMatchScheme

3.1.1 miceforest.MeanMatchScheme

Classes

`MeanMatchScheme`(mean_match_candidates, ...)

CHAPTER

FOUR

UTILS

4.1 Utility Functions

ampute_data(data[, variables, perc, ...])	Ampute Data
load_kernel(filepath[, n_threads])	Loads a kernel that was saved using save_kernel().

4.1.1 miceforest.ampute_data

miceforest.**ampute_data**(*data*, *variables=None*, *perc=0.1*, *random_state=None*)

Ampute Data

Returns a copy of data with specified variables amputed.

Parameters

- **data** (*Pandas DataFrame*) – The data to ampute
- **variables** (*None or list*) – If None, are variables are amputed.
- **perc** (*double*) – The percentage of the data to ampute.

random_state: None, int, or np.random.RandomState The random state to use.

Returns The amputed data

Return type pandas DataFrame

4.1.2 miceforest.load_kernel

miceforest.**load_kernel**(*filepath*, *n_threads=None*)

Loads a kernel that was saved using save_kernel().

Parameters

- **filepath** (*str*) – The filepath of the saved kernel
- **n_threads** (*int*) – The threads to use for decompression. By default, all threads are used.

Returns

Return type *ImputationKernel*

PYTHON MODULE INDEX

m

INDEX

T