impyte Documentation

Release 0.1.0

Andreas Rubin-Schwarz

Dec 15, 2017

Contents

Impyte is a Python module to impute missing values by prediction using machine learning algorithms.

CHAPTER 1

Introduction

One essential problem for any person dealing with data is missing values. There are several possibilities to deal with missing information, ranging from dropping data points to estimating the value based on other values in that column (i.e. average or median values). A more recent method involves machine-learning algorithms. This module offers a lightweight Python solution to calculate missing information based on the underlying relationship between data points.

The main goal of this module is to support people who are dealing with missing information to gather additional insights about the different patterns and impute them in an easy way.

There are two essential features to this module:

1. Visualization of Patterns
2. Imputation of missing information

Yet `impyte` is only one piece of the equation. In order to maximize the return in any value imputation process a deep understanding of the data is needed. As well as thorough pre-processing and cleaning of the data. Imyte takes on some of the challenges but tends to work best in concert with additional data science endeavors.

To get started with impyte is as simple as:

```
from impyte import impyte
imp = impyte.Impyter()
imp.load_data(missing_data)
imp.impute()
```

CHAPTER 2

Installation

Since this module is still in beta, you can install the latest version through its github repository via pip.

```
pip install git+git://github.com/andirs/impyte.git
```

There is also a manual way of importing the module in your project. To do so, download the repository to the folder you are performing your data work in. Afterwards you'll be able to import the `impyte` functionality through following command:

```
from impyte import impyte
```

CHAPTER 3

Requirements

The requirements are listed in `requirements.txt` and will usually be installed when proceeding through pip. When installing manually, please make sure following modules are already installed:

- Python 3.6
- sklearn 0.19
- pandas 0.21
- scipy 0.19
- pathlib 1.0.1

CHAPTER 4

API Reference

class impyte.impyte.**Impyter**(*data=None*)

Bases: object

Example usage:

```
from impyte import impyte

df = pd.read_csv("missing_values.csv")
imp = impyte.Impyter(df)

# show nan-patterns of data in one data frame
imp.pattern() # shows nan-patterns

# imputation of all single-nans using random forest
imp.impute(estimator='rf')

# imputation of all nan-patterns
imp.impute(estimator='rf', multi_nans=True)

# use f1 and r2 thresholds
imp.impute(estimator='rf', threshold={"r2": .7, "f1_macro": .7})
```

Parameters **data** (*pd.DataFrame, optional*) – Data on which to perform imputation.

The data can also be a list of lists but will be converted into a pandas DataFrame once loaded. If none, data can be loaded at a later point through `impyte.Impyter.load_data`.

Variables

- **data** (*pd.DataFrame*) – The original data, loaded by user through instantiation or `impyte.Impyter.load_data` method.
- **result** (*pd.DataFrame*) – Copy of original data on which imputation is being performed.
- **clf** (*dict*) – Holds estimator for given imputation. *(Deprecated)*

- **self.pattern_log** (*Pattern object*) – An instantiated `impyte.Pattern` object, that holds information about the NaN-pattern.
- **self.model_log** (*dict*) – Python dictionary, storing all models once `impyte.Impyter.impute` has been run
- **self.error_string** (*str*) – String representation of error messages that occured during the imputation process.
- **self.pattern_predictor_dict** (*dict*) – Python dictionary storing a pattern string and its connected list of predictors.
- **self.pattern_dependent_variable_dict** (*dict*) – Python dictionary storing a pattern string and its connected list of dependent variables.

__init__ (*data=None*)

Parameters **data** (*pd.DataFrame|list[list], optional*) – When initialized, data can be loaded directly. An alternative way is loading it with `impyte.Impyter.load_data`

static compare_features (*list_one*, *list_two*)

Compares two lists given its objects based on a comparison of Counter dicts. The order of elements is unimportant.

Parameters

- **list_one** (*list*)
- **list_two** (*list*)

Returns **True** – If list_one and list_two contain the same elements.

Return type Boolean

drop_imputation (*threshold*, *verbose=True*, *drop_pattern=False*)

Method to drop imputation results based on threshold values. Threshold values are compared against the cross-validation scores of all imputation models. If the score is lower than the threshold value, the imputation will be dropped.

An example:

```
imp = impyte.imputer(data)
imp.impute(estimator='rf')
imp.drop_imputation({"f1_macro": .8, "r2": .7})
```

Note: In the case of multi-nan, drop_imputation will average the score of all models. Yet, performing this method for multi-nan patterns is discouraged.

Further individual treatment of the data set might be more helpful in order to preprocess the information correctly. One potential action could be, to drop multi-nan columns if they contain no information.

Parameters

- **threshold** (*dict{str, float}*) – Threshold dictionary including values for r2 and f1 scores.

  An example:

```
{
    "r2" : .5,
    "f1_macro" : .7
}
```

At this point only f1 and r2 scores are being supported.

- **verbose** (*Boolean*) – Boolean flag to indicate whether results should be written to stdout.

Note: At this point there is a verbose system that distinguishes multiple layers of verbosity. This flag can also simply set to `True` in order to print out the minimum verbosity. A multi verbosity level might be enforced at a later stage.

- **drop_pattern** (*Boolean*) – Indicator if not only imputation but also pattern should be dropped.

drop_pattern (*pattern_no*, *inplace=False*)
Method to drop pattern referenced by pattern number. Drops pattern from data set and returns preliminary result. If inplace flag is set to True, internal storage of impyte object is being manipulated as well. Otherwise, a copy without the dropped pattern will be returned and the stored data set stays intact.

Parameters

- **pattern_no** (*int*)
- **inplace** (*Boolean*)

get_data ()
Returns a copy of the loaded data for quick reference.

Returns **Original Data** – A copy of the original data set can be retrieved through this method.

Return type pd.DataFrame

get_model (*pattern_no*)
Returns model that matches pattern number.

Parameters **pattern_no** (*int*) – Pattern number to receive fitting model.

Returns **model**

Return type ImpyterModellImpyterMultiModel

get_pattern (*pattern_no*, *result=False*)
Returns data points for a specific pattern_no for further investigation.

Parameters

- **pattern_no** (*int*) – Index value that indicates pattern
- **result** (*Boolean*) – Flag to show if original or result data should be sliced.

Returns **data** – Data points that have a certain pattern, if `result` is set to `True` the data is result data, otherwise a slice of the original data is being returned.

Return type pd.DataFrame

get_result ()
Returns a copy of the result data for reference.

Returns **Result Data** – A copy of the result data.

Return type pd.DataFrame

get_summary (*importance_filter=True*)
Shows simple overview of missing values. Returns table with information on missing values per column, its percentage and the count of unique values within that column.

Setting the importance filter flag to True shows only columns that have some missing values. This is helpful for data sets with a large amount of variables and only few nan-values.

Parameters **importance_filter** (*Boolean*) – Show only features with at least one missing value.

Returns **Summary table**

Return type pd.DataFrame

impute (*data=None*, *cv=5*, *verbose=True*, *estimator='rf'*, *multi_nans=False*, *one_hot_encode=True*, *auto_scale=True*, *threshold={'f1_macro': None*, *'r2': None}*, *recompute=False*)

Impute is the core method of impyte. The method works out of the box and uses Random Forest estimators per default to impute missing values. It automatically performs cross-validation to showcase the potential accuracy of the imputation.

Scoring that is being used is f1_macro score for classifiers (supporting binary and multi-class) and r2 for regression models.

In order to fill in only columns that surpass a certain scoring threshold (i.e. f1 score > .7), the threshold parameter can be set. The threshold values are being transmitted through a dictionary.

Note: Multi Nans

Prediction of values with multi-nan is a last resort option. This might be suitable for certain edge cases but if the score values are low it should be considered dropping the feature or the data points all together.

Parameters

- **data** (*pd.DataFrame*) – Data to be imputed.
- **cv** (*int*) – Amount of cross-validation runs.
- **verbose** (*Boolean*) – Indicator, whether prediction results should be printed out.
- **estimator** (*str*) – Estimators can be chosen through a simple string abbreviation. This table outlines the potential options.

Abbreviation	Estimator
'rf'	Random Forest
'svm'	Support Vector Machine
'sgd'	Stochastic Gradient Descent
'knn'	KNearest Neighbor
'bayes'	(Naive) Bayes
'dt'	Decision Tree
'gb'	Gradient Boosting

- **multi_nans** (*Boolean*) – Indicator if data points with multiple NaN values should be imputed as well
- **one_hot_encode** (*Boolean*) – If set to True one-hot-encoding of categorical variables happens
- **auto_scale** (*Boolean*) – If set to True continuous variables are automatically scaled and transformed back after imputation.
- **threshold** (*dict{str, float}*) – Classification and regression threshold cut-offs. At this point f1 score and R2.

- **recompute** (*Boolean*) – Indicator whether the system should recompute the imputation or use stored models if possible.

Note: Impyte will print a warning to the stdout if the data set might contain too few rows in general to properly compute any imputation method.

`load_data`(*data*)

Function to load data into Impyter class. Requires a pandas DataFrame to load. Otherwise, the input is being transformed into a DataFrame. While loading the data is being copied into the object, to stay clear of consistency issues with the original data set.

Parameters **`data`** – preferably pandas DataFrame

`load_model`(*filename*, *path='models/'*)

Load a stored machine learning model to perform value imputation.

Parameters

- **filename** (*str*) – Filename of model
- **path** (*str*) – Path to model (default value is 'models/')

`map_model_to_pattern`(*mdl*)

Checks model for similarity to stored patterns and returns pattern number if a match is found.

Parameters **mdl** (*ImpyterModel*)

Returns **pattern_no** – If no pattern number can be found, a None value will be returned.

Return type int

`map_multimodel_to_pattern`(*mmdl*)

Checks multi-model for similarity to stored patterns and returns pattern number if a match is found.

Parameters **mmdl** (*ImpyterMultiModel*)

Returns **pattern_no** – If no pattern number can be found, a None value will be returned.

Return type int

`one_hot_decode`(*data*)

Decodes one-hot-encoded features into single column again. Generally speaking, this function inverses the one-hot-encode function.

Parameters **data** (*pd.DataFrame*) – DataFrame that has one-hot-encoded columns processed by `impyte.Impyter.one_hot_encode`.

Returns **Data set** – Data set with collapsed information.

Return type pd.DataFrame

`one_hot_encode`(*data*, *verbose=False*)

Uses pandas get_dummies method to return a one-hot-encoded DataFrame.

Parameters

- **data** (*pd.DataFrame*)
- **verbose** (*Boolean*)

Returns DataFrame with one-hot-encoded categorical values.

Return type Data set - pd.DataFrame

pattern(*recompute=False*)

Returns missing value patterns of data set. Leverages `impyte.Pattern._compute_pattern` and *impyte.Pattern.get_pattern* methods to compute and return an overview of all existant NaN patterns in the data set. The overview shows a NaN in the column where a data point was missing and 1 for all complete slots. On the right hand side is a count variable to indicate how often that pattern was found. The patterns are always sorted by count and it is not given, that pattern 0 is always the pattern with only complete cases.

A potential result table could look like this, where `NaN` indicates the column contains missing values in this pattern. The `Count` column shows how many observations of this NaN-pattern are in the data set.

Pattern	left_socks	right_socks	Count
0	1	1	15
1	NaN	1	6
2	1	NaN	6
3	NaN	NaN	4

For additional information (and a rather sad joke) please head over to `impyte.Pattern`.

Parameters **recompute** (*Boolean*) – Flag to indicate whether patterns should be recomputed from the original data set. This is an important feature if for example a pattern has been dropped and should be incorporated again.

Returns **NaN-Pattern Table** – Table with overview of NaN-patterns.

Return type pd.DataFrame

save_model(*pattern_no=None*, *filename=None*, *path='models/'*)

Stores an imputation model for either the whole data set or a particular pattern in a pickle file. If pattern_no is not set, the method stores all models. If filename is not set, an automated name is being produced including a timestamp.

Parameters

- **pattern_no** (*int, optional*) – Pattern number that points to a certain NaN-Pattern model which in turn references a `impyte.ImpyteModel` or `impyte.ImpyteMultiModel`.
- **filename** (*str, optional*) – If value is not set, an automated name is being created.
- **path** (*str*) – (default value is 'models/' which will automatically create a model for that)

set_unique(*unique_no*)

Set unique values for imputation.

Parameters **unique_no** (*int*) – Positive number that indicates a threshold for unique values needed in a column for it to be counted as continuous variable.

class impyte.impyte.**ImpyterModel**(*estimator_name*, *model=None*, *pattern_no=None*, *feature_name=None*, *scores=None*, *scoring=None*, *predictor_variables=None*, *pattern_string=None*, *y_scaler=None*)

Bases: `object`

Stores computed Impyter machine learning models and relevant information that is linked to the model and pattern.

Variables

- **model** (*sklearn Machine Learning Model*) – Contains a trained machine learning model for given imputation task.
- **pattern_no** (*int*) – Indicator for pattern number.

- **feature_name** (*str|int*) – Name of the dependent variable.
- **scores** (*list*) – List of all cross-validation scores. The average of this list is being used as the threshold score.
- **estimator_name** (*str*) – String representation of the Machine Learning model.
- **scoring** (*str*) – String representation of the scoring measurement ('r2' or 'f1_macro' right now)
- **predictor_variables** (*list*) – Contains names of all independent variables used for the imputation task.
- **pattern_string** (*tuple*) – Tuple representation of pattern string. Can be used for identification of patterns.
- **y_scaler** (*sklearn.preprocessing.StandardScaler object*) – StandardScaler object that contains additional information in case the model was used with `auto_scale = True`.

__init__(*estimator_name*, *model=None*, *pattern_no=None*, *feature_name=None*, *scores=None*, *scoring=None*, *predictor_variables=None*, *pattern_string=None*, *y_scaler=None*)

Parameters

- **estimator_name** (*str*) – Name of machine learning model
- **model** (*sklearn Machine Learning Model*) – Sklearn machine learning estimator object
- **pattern_no** (*int*) – Pattern number associated with nan-pattern.
- **feature_name** (*str|int*) – Name of dependent variable.
- **scores** (*list[float]*) – Collection of all cross-validation scores.
- **scoring** (*str*) – String representation of scoring function. (i.e. "r2" or "f1_macro")
- **predictor_variables** (*list[str|int]*) – List of names of all independent variables.
- **pattern_string** (*tuple*) – Tuple representation of a certain pattern.
- **y_scaler** (*sklearn.preprocessing.StandardScaler object*) – StandardScaler object that contains additional information in case the model was used with `auto_scale = True`.

class impyte.impyte.**ImpyterMultiModel**(*pattern_string*)

Bases: `object`

Stores multi-nan imputations in the form of a list of `impyte.ImpyterModel` objects.

Variables

- **_model_list** (*list*) – Collection of all ImpyterModel that are needed to compute the given multi-nan pattern.
- **count** (*int*) – Amount of models that are stored in ImpyterModels.
- **pattern_string** (*tuple*) – Tuple representation of multi-nan pattern.

__init__(*pattern_string*)

Parameters **pattern_string** (*tuple*) – References a pattern by tuple.

append(*model*)

Appends an additional ImpyterModel object to the list of models.

Parameters **model** (*ImpyterModel object*) – The model to be appended to the model list

static check_and_append(*input_list*, *storage_list*)
Extension helper method to append items to a pre-existing list if not included.

Parameters

- **input_list** (*list*) – List with items to append.
- **storage_list** (*list*) – List that serves as storage item for all items.

Returns **storage_list** – Collection of all unique elements from input_list and storage_list

Return type list

static combine_in_list(*input_list*, **args*)
Extension helper method to add multiple and single arguments to a pre-existing list.

Parameters

- **input_list** (*list*) – Pre-existing list.
- **args** (*list*) – List or single values to be extended to list.

Returns **extended input_list**

Return type list

get_dependend_and_independent_variables()
For all models stored in the object, collect their dependent and independent variables.

As an example, if we had a multi-nan model that stored two ImpyterModels to predict `right_socks` and `left_socks`, the variables stored in the response would look like this:

```
{
    "independent_variables": ["time_of_year", "pants", "hat"],
    "dependent_variables":   ["right_socks", "left_socks"]
}
```

Returns **Variables** – Dictionary including independent and dependent variables. Can be accessed through "independent_variables" and "dependent_variables".

Return type dict{str, list}

class impyte.impyte.**NanChecker**
Bases: object

Class that checks data set, lists or single values for NaN occurrence.

Examples

Testing list for NaN values:

```
nan_array = ["Test", None, '', 23, [None, "42"]]
nan_checker = impyte.NanChecker()
print(nan_checker.is_nan(nan_array))
>>> [False, True, True, False, [True, False]]
```

static is_nan(*data*, *nan_vals=None*, *recursive=True*)
Detect missing values (NaN in numeric arrays, empty strings in string arrays).

Parameters

- **data** (*{numpy.ndarray|str|list|int|float}*) – Data to be investigated for NaN values.

- **nan_vals** (*list*) – Array of values that count as NaN values - if empty, "" and None are being used
- **recursive** (*boolean*) – Flag that determines whether the lists should be handled in recursive manner

Returns **result** – Array or bool indicating whether an object is null or if an array is given which of the element is null.

Return type Boolean

class impyte.impyte.**Pattern**(*unique_instances=10*)

Bases: object

Class that calculates, stores and visualizes NaN patterns and their indices.

Variables

- **column_names** (*list*) – Python list storing names of all columns that are in data set.
- **complete_idx** (*int*) – Integer containing pattern number with only complete cases
- **continuous_variables** (*list*) – Python list containing column names of all continuous variables. (i.e. columns that contain values in a range from 0.0 to 1.0)
- **discrete_variables** (*list*) – Python list containing column names of all discrete variables. (i.e. columns that contain values such as "red", "blue", "green")
- **easy_access** (*dict{tuple, list}*) – Python dictionary holding NaN-pattern strings and mapping them to a list of the names of columns that contain NaN values in the given NaN-pattern.

  As an example:

```
{
    ('NaN', 1):     ['left_socks'],
    (1, 'NaN'):     ['right_socks'],
    ('NaN', 'NaN'): ['left_socks', 'right_socks']
}
```

- **missing_per_column** (*list*) – Python list used to store summarization results, to make the use of impyte.Pattern.get_missing_value_percentage more efficient (the default is None)
- **nan_checker** (*NanChecker object*) – An instantiated impyte.NanChecker object, that can be used to analyze values and rows regarding their NaN values.
- **pattern_index_store** (*dict{int, list}*) – Python dictionary holding a list of indices for every pattern number. This dictionary is being used to look up the corresponding data points in a pandas DataFrame.

  As an example:

```
{
    0: [0, 1, 2, 3, 4],     # pattern_number: indices
    1: [5, 6, 7, 8, 9]
}
```

  This pattern log consists out of 2 patterns (0 and 1) each pointing to 5 indices.

- **pattern_store** (*dict{str, pd.DataFrame}*) – Python dictionary storing the pattern table. The table (in pd.DataFrame form) can be accessed by self.pattern_store['result'].

A potential result table could look like this, where `NaN` indicates the column contains missing values in this pattern. The `Count` column shows how many observations of this NaN-pattern are in the data set.

Pattern	left_socks	right_socks	Count
0	1	1	15
1	NaN	1	6
2	1	NaN	6
3	NaN	NaN	4

Let's hope these left and right socks are of the same color at least...

- **result_pattern** (*dict{tuple, int}*) – Python dictionary version of pattern counts. Makes computation and alterations easier.
- **tuple_counter** (*int*) – Value storing the amount of different patterns after performing pattern analysis. (the default is 0)
- **tuple_counter_dict** (*dict*) – Python dictionary mapping pattern strings to pattern number.
- **tuple_dict** (*dict{tuple, int}*) – As an example:

```
{
    ('NaN', 1):     1,  # points to pattern 1
    (1, 'NaN'):     2,
    ('NaN', 'NaN'): 3
}
```

- **unique_instances** (*int*) – Value indicating the minimum value for a column of unique values to be considered as continuous variable when having the proper dtype

  (the default is 10, which implies that columns with over 10 unique values are being labeled as continuous variables if containing numbers).
- **pattern_predictor_dict** (*dict*) – Python dictionary mapping pattern strings to their independent variable names.
- **pattern_dependent_dict** (*dict*) – Python dictionary mapping pattern string to their dependent variable names.

__init__(*unique_instances=10*)

When instantiating a `impyte.Pattern` object, most values are being initialized as being empty or None.

Parameters **unique_instances** (*int*) – Value indicating the minimum value for a column of unique values to be considered as continuous variable when having the proper dtype

(the default is 10, which implies that columns with over 10 unique values are being labeled as continuous variables if containing decimal numbers).

get_column_name(*patter_no*)

Returns the column name(s) that contain missing information of a certain NaN-pattern.

Parameters **patter_no** (*int*) – Number or identifier of pattern

Returns **Column names** – If patter_no has been computed, a list of all column names associated with pattern_no are being returned.

Return type list

get_complete_id()
Returns pattern number of observations that don't contain any missing information.

Returns **Pattern number**

Return type int

get_complete_indices()
Function to determine complete cases based on results table. Leverages pre-computed information and is quicker than pandas dropna method.

Returns **Indices** – List of indices that point to rows with complete cases

Return type list

get_continuous()
Returns copy of continuous variable names.

Returns **Continuous variable names**

Return type list

get_discrete()
Returns copy of discrete variable names.

Returns **Discrete variable names**

Return type list

get_missing_value_percentage(*data*, *importance_filter=False*)
Combines information regarding the values in the data set and returns them in a concise way.

A potential summary table could look like this.

Column	Complete	Missing	Percentage	Unique
left_socks	21	6	19.4 %	2
right_socks	21	6	19.4 %	2

Parameters

- **data** (*pd.DataFrame*) – data refers to the information the user wants to analyze (Usually the result data set stored in `Impyte.impyter`)
- **importance_filter** (*Boolean*) – Flag, to don't show columns that have no missing values. This might make sense for data sets with a lot of columns that have no missing values.

  (default value is False, stating that all columns are important)

Returns **Summary table** – Contains information regarding complete, missing and unique values in the data set.

Return type pd.DataFrame

get_multi_nan_pattern_nos(*multi=True*)
Returns all pattern numbers of multi-nans or single-nans

Parameters **multi** (*Boolean*) – Flag indicating whether the user wants to retrieve multi or single-nan pattern numbers.

Returns **Pattern Numbers** – All single or multi-nan pattern numbers.

Return type list

get_pattern(*data=None*, *recompute=False*)
Returns NaN-patterns based on primary computation or initiates new computation of NaN-patterns.

Parameters

- **data** (*pd.DataFrame*)
- **recompute** (*Boolean*) – If set True, stored results are being disregarded

Returns **Pattern overview** – Table representation of all NaN-patterns and their counts.

Return type pd.DataFrame

get_pattern_indices(*pattern_no*)
Returns data points for a specific pattern_no for further investigation.

Parameters **pattern_no** (*int*) – Index value that indicates pattern number.

Returns **Indices** – Indices that correspond to a pattern number.

Return type list

get_single_nan_pattern_nos()
Returns all pattern numbers that contain only single nans.

Returns **Pattern Numbers** – All single pattern numbers containing single-nans.

Return type list

remove_pattern(*pattern_no*)
Removes a certain pattern. Deletes dictionary entry in the pattern index store as well as drops the entry in the results table.

Parameters **pattern_no** (*int*) – Index value that indicates pattern.

CHAPTER 5

Help

5.1 FAQs

Below are some pointers towards the right direction if something breaks. If you encounter any other error please feel free to reach out.

When imputing my estimator raises ValueError: Unknown label type: 'continuous'

> **Hint:** This might happen, if there is too little information for impyte to correctly distinguish your data type. This error essentially means, you're handing a continuous data type [i.e. a float] to a classifier which expects a class or discrete value.
>
> To solve this problem, you can set the unique value threshold to a lower value. (standard value is 10 unique instances).

5.2 Index

The index stores an alphabetical list of the API reference.

- genindex

5.3 License

5.4 Contact

Feel free to contact me here or add me on linkedin.

Python Module Index

i

Index

Symbols

A

C

D

G

I

L

M

N

O

P

R

S