ampute

Release 0.1.0.dev0

G. Lemaitre

Nov 28, 2021

CONTENTS

Date: Nov 28, 2021 **Version**: 0.1.0.dev0

Useful links: Binary Installers | Source Repository | Issues & Ideas

Ampute is a open source (new BSD) Python package that provides some `scikit-learn` compatible utilities to generate dataset with missing values from a given dataset.

CHAPTER

ONE

GETTING STARTED

1.1 Prerequisites

The `ampute` package requires the following dependencies:

- python (>=3.7)
- numpy (>=1.17.5)
- scipy (>=1.0.1)
- scikit-learn (>=1.0)

1.2 Install

1.2.1 From PyPi or conda-forge repositories

`ampute` is currently available on the PyPi's repositories and you can install it via `pip`:

```
pip install -U ampute
```

The package is release also in `conda-forge` platform:

```
conda install -c conda-forge ampute
```

1.2.2 From source available on GitHub

If you prefer, you can clone it and run the setup.py file. Use the following commands to get a copy from Github and install all dependencies:

```
git clone https://github.com/glemaitre/ampute.git
cd ampute
pip install .
```

Be aware that you can install in developer mode with:

```
pip install --no-build-isolation --editable .
```

If you wish to make pull-requests on GitHub, we advise you to install pre-commit:

```
pip install pre-commit
pre-commit install
```

1.3 Test and coverage

You want to test the code before to install:

```
$ make test
```

You wish to test the coverage of your version:

```
$ make coverage
```

You can also use `pytest`:

```
$ pytest ampute -v
```

1.4 Contribute

You can contribute to this code through Pull Request on GitHub. Please, make sure that your code is coming with unit tests to ensure full coverage and continuous integration in the API.

CHAPTER

TWO

USER GUIDE

Learning a predictive model in presence of missing data is a difficult task. This topic is still under investigation in research. In this regard, these investigations required to study different type of missingness mechanisms in a control environment.

The `ampute` package provides a set of tools to amputate a given dataset. It allows to handcraft the missingness mechanism and pattern that later be studied.

2.1 What are missing values?

Let's first define a couple of notation that we will use in this documentation Our dataset is denoted by X of shape (n, p) where n is the number of samples and p is the number of features. Programmatically, we can represent such a matrix as a NumPy array:

```
>>> from numpy.random import default_rng
>>> rng = default_rng(0)
>>> n_samples, n_features = 5, 3
>>> X = rng.normal(size=(n_samples, n_features))
>>> X
array([[ 0.12573022, -0.13210486,  0.64042265],
       [ 0.10490012, -0.53566937,  0.36159505],
       [ 1.30400005,  0.94708096, -0.70373524],
       [-1.26542147, -0.62327446,  0.04132598],
       [-2.32503077, -0.21879166, -1.24591095]])
```

If we are lucky, for each sample (i.e. row of X), we always have an observation. However, we could be unlucky -e.g. one of the sensor collecting data was broken- and some observations could be missing.

In NumPy, the sentinel generally used to represent missing values is `np.nan`. For instance, if for the first sample in X, the value of the second feature is not collected, we should have:

```
>>> import numpy as np
>>> X_missing = X.copy()
>>> X_missing[0, 1] = np.nan
>>> X_missing
array([[ 0.12573022,         nan,  0.64042265],
       [ 0.10490012, -0.53566937,  0.36159505],
       [ 1.30400005,  0.94708096, -0.70373524],
       [-1.26542147, -0.62327446,  0.04132598],
       [-2.32503077, -0.21879166, -1.24591095]])
```

In a machine learning setting where one is interested about training a predictive model, such missing values are problematic. Let's define a linear relationship (with some noise) between the data matrix and the target vector that we want to predict:

```
>>> coef = rng.normal(size=n_features)
>>> y = X @ coef + rng.normal(scale=0.1, size=n_samples)
>>> y
array([-0.18157161,  0.2046067 , -1.26059589,  1.38942449,  2.14918582])
```

If we would like to train a linear regression on this problem, we could use for instance `scikit-learn`:

```
>>> from sklearn.linear_model import LinearRegression
>>> reg = LinearRegression().fit(X, y)
>>> reg.coef_
array([-0.67735802, -0.70333477, -0.32484547])
```

However, with the presence of missing values, we cannot use this approach:

```
>>> try:
...     reg.fit(X_missing, y)
... except ValueError as e:
...     print(e)
Input X contains NaN.
LinearRegression does not accept missing values encoded as NaN natively.
For supervised learning, you might want to consider
sklearn.ensemble.HistGradientBoostingClassifier and Regressor which accept
missing values encoded as NaNs natively. Alternatively, it is possible to
preprocess the data, for instance by using an imputer transformer in a pipeline
or drop samples with missing values.
See https://scikit-learn.org/stable/modules/impute.html
```

`scikit-learn` is giving you a straightforward answer informing you that it does not accept missing values. Thus, one needs to a strategy to deal with missing values.

In the following section, we describe what are the missingness mechanisms used in the literature to simulate the presence of missing values in a dataset.

2.2 Missingness mechanisms

In the literature, there are three reported mechanism to control the missingness of a dataset: missing completely at random (MCAR), missing at random (MAR), and missing not at random (MNAR). We will give a brief description of each of them.

MCAR strategy is straightforward. For a given feature, missing values are created at random. Let's imagine the first feature in X contains missing values. MCAR strategy is be equivalent to:

```
>>> X_missing_mcar = X.copy()
>>> missing_values_indices = rng.choice(
...     n_samples, size=n_samples // 2, replace=False
... )
>>> X_missing_mcar[missing_values_indices, 0] = np.nan
>>> X_missing_mcar
array([[ 0.12573022, -0.13210486,  0.64042265],
```

(continues on next page)

(continued from previous page)

```
       [ 0.10490012, -0.53566937,  0.36159505],
       [        nan,  0.94708096, -0.70373524],
       [        nan, -0.62327446,  0.04132598],
       [-2.32503077, -0.21879166, -1.24591095]])
```

Here, there is not link between the missing values and the features in X.

CHAPTER

THREE

API REFERENCE

This is the full API documentation of the `ampute` toolbox.

3.1 Univariate amputation

UnivariateAmputer([strategy, subset, ...])	Ampute a datasets in an univariate manner.

3.1.1 UnivariateAmputer

class ampute.**UnivariateAmputer**(*strategy='mcar'*, *subset=None*, *ratio_missingness=0.5*, *copy=True*, *random_state=None*)

Ampute a datasets in an univariate manner.

Univariate imputation refer to the introduction of missing values, one feature at a time.

Parameters

strategy [str, default="mcar"] The missingness strategy to ampute the data. Possible choices are:

- `"mcar"`: missing completely at random. This strategy implies that the missing values are amputated for a feature without any dependency with other features.

subset [list of {int, str}, int or float, default=None] The subset of the features to be amputated. The possible choices are:

- `None`: all features are amputated.
- `list of {int, str}`: the indices or names of the features to be amputated.
- `float`: the ratio of features to be amputated.
- `int`: the number of features to be amputated.

ratio_missingness [float or array-like, default=0.5] The ratio representing the amount of missing data to be generated. If a `float`, all features to be imputed will have the same ratio. If an array-like, the ratio of missingness for each feature will be drawn from the array. It should be consistent with `subset` when a list is provided for `subset`.

copy [bool, default=True] Whether to perform the amputation inplace or to trigger a copy. The default will trigger a copy.

Attributes

amputated_features_indices_ [ndarray of shape (n_selected_features,)] The indices of the features that have been amputated.

Examples

```
>>> from numpy.random import default_rng
>>> rng = default_rng(0)
>>> n_samples, n_features = 5, 3
>>> X = rng.normal(size=(n_samples, n_features))
```

One can amputate values using the common transformer `scikit-learn` API:

```
>>> amputer = UnivariateAmputer(random_state=42)
>>> amputer.fit_transform(X)
array([[ 0.12573022, -0.13210486,  0.64042265],
       [        nan, -0.53566937,         nan],
       [        nan,         nan,         nan],
       [        nan,         nan,  0.04132598],
       [-2.32503077,         nan, -1.24591095]])
```

The amputer can be used in a scikit-learn `Pipeline`.

```
>>> from sklearn.impute import SimpleImputer
>>> from sklearn.pipeline import make_pipeline
>>> pipeline = make_pipeline(
...     UnivariateAmputer(random_state=42),
...     SimpleImputer(strategy="mean"),
... )
>>> pipeline.fit_transform(X)
array([[ 0.12573022, -0.13210486,  0.64042265],
       [-1.09965028, -0.53566937, -0.18805411],
       [-1.09965028, -0.33388712, -0.18805411],
       [-1.09965028, -0.33388712,  0.04132598],
       [-2.32503077, -0.33388712, -1.24591095]])
```

You can use the class as a callable if you don't need to use a `sklearn.pipeline.Pipeline`:

```
>>> from ampute import UnivariateAmputer
>>> UnivariateAmputer(random_state=42)(X)
array([[ 0.12573022, -0.13210486,  0.64042265],
    [        nan, -0.53566937,         nan],
    [        nan,         nan,         nan],
    [        nan,         nan,  0.04132598],
    [-2.32503077,         nan, -1.24591095]])
```

Methods

fit(X[, y])	Validation of the parameters of amputer.
fit_transform(X[, y])	Fit to data, then transform it.
get_params([deep])	Get parameters for this estimator.
set_params(**params)	Set the parameters of this estimator.
transform(X[, y])	Amputate the dataset `X` with missing values.

`fit`(*X*, *y=None*)
Validation of the parameters of amputer.

Parameters

X [{array-like, sparse matrix, dataframe} of shape (n_samples, n_features)] The dataset to be amputated.

y [Ignored] Present to follow the scikit-learn API.

Returns

self The validated amputer.

`fit_transform`(*X*, *y=None*, ***fit_params*)
Fit to data, then transform it.

Fits transformer to `X` and `y` with optional parameters `fit_params` and returns a transformed version of `X`.

Parameters

X [array-like of shape (n_samples, n_features)] Input samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] Target values (None for unsupervised transformations).

****fit_params** [dict] Additional fit parameters.

Returns

X_new [ndarray array of shape (n_samples, n_features_new)] Transformed array.

`get_params`(*deep=True*)
Get parameters for this estimator.

Parameters

deep [bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [dict] Parameter names mapped to their values.

`set_params`(***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params** [dict] Estimator parameters.

Returns

self [estimator instance] Estimator instance.

`transform`(*X*, *y=None*)
Amputate the dataset `X` with missing values.

Parameters

X [{array-like, sparse matrix, dataframe} of shape (n_samples, n_features)] The dataset to be amputated.

y [Ignored] Present to follow the scikit-learn API.

Returns

X_amputed [{ndarray, sparse matrix, dataframe} of shape (n_samples, n_features)] The dataset with missing values.

Examples using `ampute.UnivariateAmputer`

- *Univariate amputation*

CHAPTER

FOUR

EXAMPLES

General-purpose and introductory examples for the `ampute` package.

4.1 Univariate amputation

This example demonstrates different ways to amputate a dataset in an univariate manner.

```
# Author: G. Lemaitre
# License: BSD 3 clause
```

```
import sklearn
import seaborn as sns

sklearn.set_config(display="diagram")
sns.set_context("poster")
```

Let's create a synthetic dataset composed of 10 features. The idea will be to amputate some of the observations with different strategies.

```
import pandas as pd
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=10_000, n_features=10, random_state=42)

feature_names = [f"Features #{i}" for i in range(X.shape[1])]
X = pd.DataFrame(X, columns=feature_names)
```

4.1.1 Missing completely at random (MCAR)

We will show how to amputate the dataset using a MCAR strategy. Thus, we will amputate 3 given features randomly selected.

```
import numpy as np

rng = np.random.default_rng(42)
n_features_with_missing_values = 3
features_with_missing_values = rng.choice(
    feature_names, size=n_features_with_missing_values, replace=False
)
```

Now that we selected the features to be amputated, we can create an transformer that can amputate the dataset.

```
from ampute import UnivariateAmputer

amputer = UnivariateAmputer(
    strategy="mcar",
    subset=features_with_missing_values,
    ratio_missingness=[0.2, 0.3, 0.4],
)
```

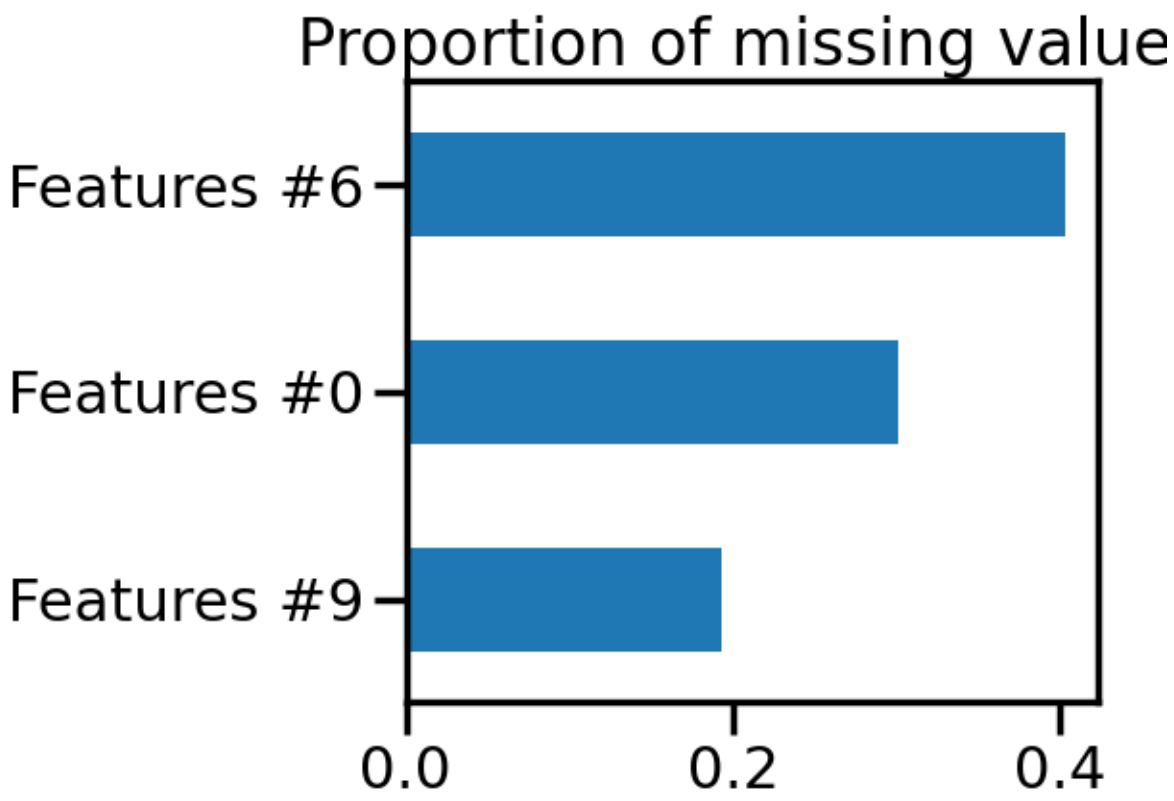
If we want to amputate the full-dataset, we can directly use the instance of *UnivariateAmputer* as a callable.

```
X_missing = amputer(X)
X_missing.head()
```

We can quickly check if we get the expected amount of missing values for the amputated features.

```
import matplotlib.pyplot as plt

ax = X_missing[features_with_missing_values].isna().mean().plot.barh()
ax.set_title("Proportion of missing values")
plt.tight_layout()
```



Thus we see that we have the expected amount of missing values for the selected features.

Now, we can show how to amputate a dataset as part of the `scikit-learn` `Pipeline`.

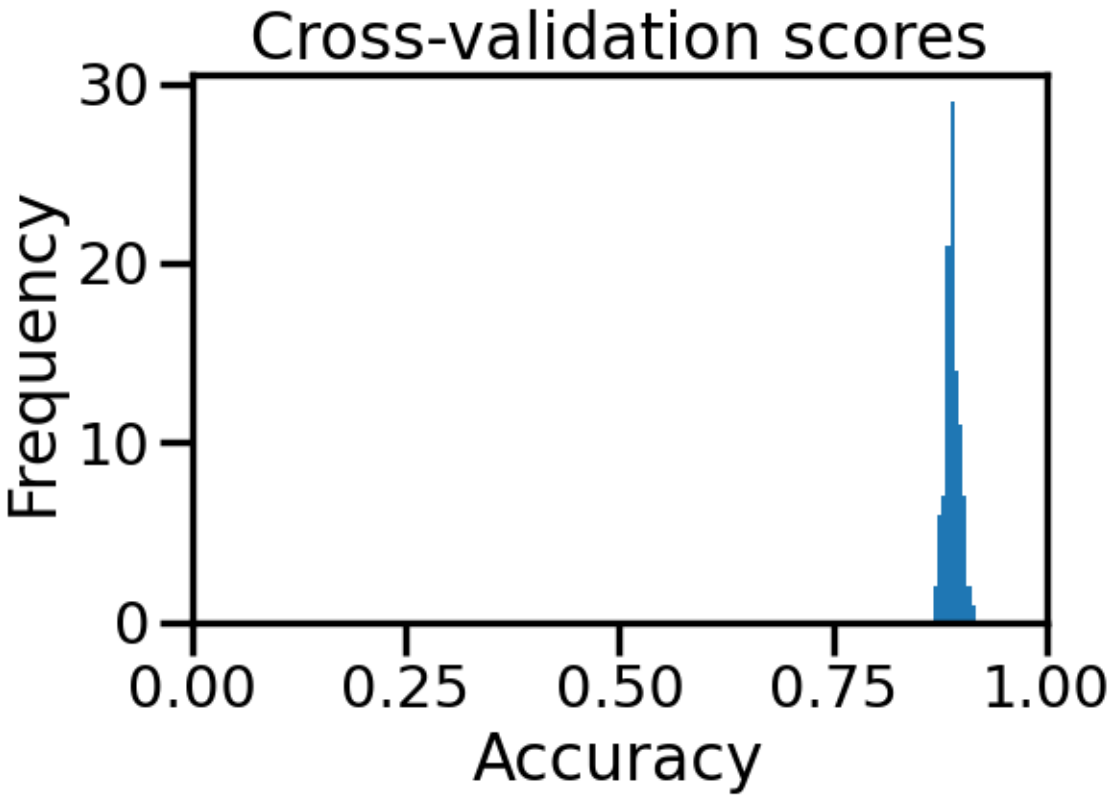
```
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import ShuffleSplit

model = make_pipeline(
    amputer,
    StandardScaler(),
    SimpleImputer(strategy="mean"),
    LogisticRegression(),
)
model
```

Now that we have our pipeline, we can evaluate it as usual with any cross-validation tools provided by `scikit-learn`.

```
n_folds = 100
cv = ShuffleSplit(n_splits=n_folds, random_state=42)
results = pd.Series(
    cross_val_score(model, X, y, cv=cv, n_jobs=2),
    index=[f"Fold #{i}" for i in range(n_folds)],
)
```

```
ax = results.plot.hist()
ax.set_xlim([0, 1])
ax.set_xlabel("Accuracy")
ax.set_title("Cross-validation scores")
plt.tight_layout()
```



Total running time of the script: (0 minutes 4.324 seconds)

CHAPTER

FIVE

RELEASE HISTORY

5.1 Version 0.1.0 (under development)

5.1.1 Changelog

New features

Enhancements

Bug fixes

Maintenance

Deprecation

CHAPTER

SIX

ABOUT US

6.1 History

6.1.1 Development lead

This project started during a rainy Sunday afternoon by G. Lemaitre that was somehow pretty bored.

6.2 Contributors

Refers to GitHub contributors page.

INDEX