Photo by Gabriel Crismariu on Unsplash

Coding a custom imputer in scikit-learn

Learn how to create custom imputers, including groupby aggregation for more advanced use-cases

Eryk Lewinson May 21, 2020 · 5 min read ★

imputer from the `sklearn.impute` module). However, often the simplest approach might not be the best one and we could gain some extra performance by using a more sophisticated approach.

That is why in this article I wanted to demonstrate how to code a custom `scikit-learn` based imputer. To make the case more interesting, the imputer will fill in the missing values based on the groups' averages/medians.

Why should you write custom imputer as classes?

Before jumping straight into coding I wanted to elaborate on a few potential reasons why writing a custom imputer class (inheriting from `scikit-learn`) might be worth your time:

- It can help you with developing your programming skills — while writing imputers inheriting from `scikit-learn` you learn about some best practices already used by the contributors. Additionally, via inheritance you can use some of the already prepared methods. This way, your code will be better/cleaner and potentially more robust to some unforeseen issues.
- Your custom classes can be further developed over time and potentially shared with other users (or maybe even integrated into `scikit-learn`!)
- More on the practical side, by creating imputers using the `scikit-learn` framework you make them compatible with `scikit-learn`'s `Pipelines`, which make the project's flow much cleaner and easier to reproduce/productionize. Another practical matter is the clear distinction between the `fit` and `transform` methods, so you will not accidentally introduce **data leakage** — including the test data in the process of determining the values to be used for imputing.

Implementing the custom imputer

In this section, we will implement the custom imputer in Python.

Setup

First, we load all the required libraries:

```
from sklearn.utils.validation import check_is_fitted
```

custom_imputer_1.py hosted with ♡ by GitHub view raw

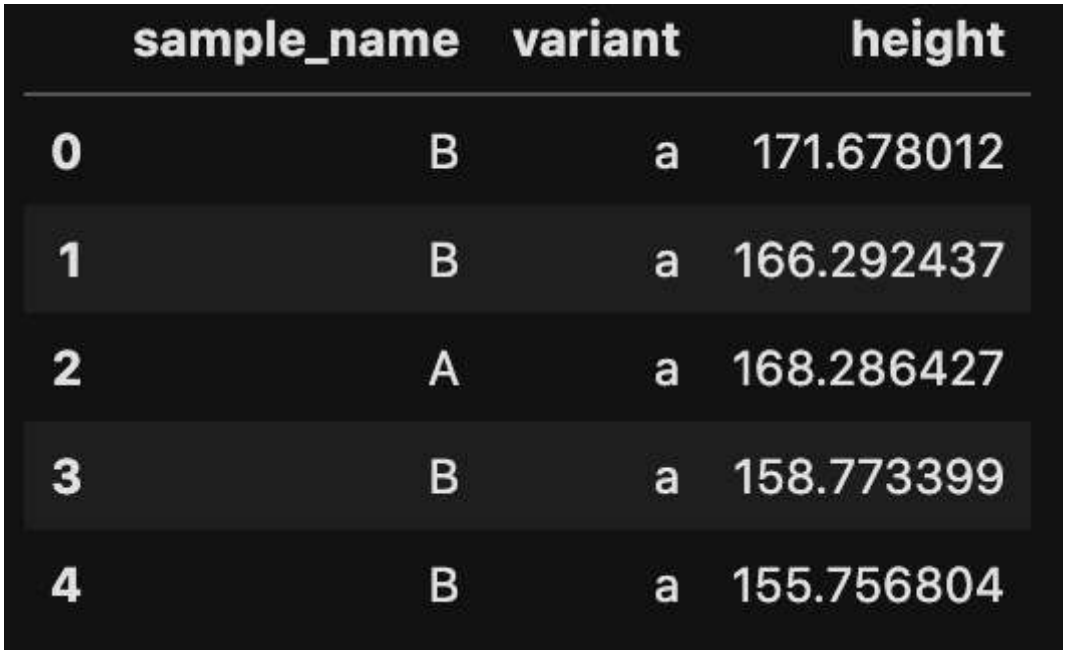
For writing this article, I used `scikit-learn` version 0.22.2.

Generating sample data

For this article we will use a toy dataset. We assume the case of collecting the height of people coming from two different populations (samples `A` and `B`), hence some variability in the data. Additionally, the first sample also has a distinguishing feature called `variant` (with values of `a` and `b`). What is behind this naming structure is of no importance, the goal was to have two different levels of possible aggregation. Then, we sample the heights from the Normal distribution (using `numpy.random.normal`) with different values of the scale and location parameters per `sample_name`.

```
# setting the seed
np.random.seed(42)

# generating the two samples
sample_a = pd.DataFrame({'sample_name': 'A',
                         'variant': np.random.choice(['a', 'b'], size=100),
                         'height': np.random.normal(loc=170, scale=5, size=100)})

sample_b = pd.DataFrame({'sample_name': 'B',
                         'variant': 'a',
                         'height': np.random.normal(loc=165, scale=7, size=100)})

# concatenating the samples
df = pd.concat([sample_a, sample_b], axis=0, ignore_index=True) \
       .sample(frac=1) \
       .reset_index(drop=True)

# preview the data
df.head()
```

custom_imputer_2.py hosted with ♡ by GitHub view raw



	sample_name	variant	height
0	B	a	171.678012
1	B	a	166.292437
2	A	a	168.286427
3	B	a	158.773399
4	B	a	155.756804

Preview of the generated data

Then, we replace 10 random heights with NaN values using the following code:

```
ind_to_replace = np.random.choice(range(len(df)), 10, replace=False)
df.loc[ind_to_replace, 'height'] = np.nan
```

custom_imputer_3.py hosted with ♡ by GitHub view raw

Now, the `DataFrame` is ready for imputation.

Coding the imputer

It is time to code the imputer. You can find the definition of the class below:

```
class GroupImputer(BaseEstimator, TransformerMixin):
    '''
    Class used for imputing missing values in a pd.DataFrame using either mean or median of a gr

    Parameters
    ----------
    group_cols : list
        List of columns used for calculating the aggregated value
    target : str
```

```
13
14      Returns
15      -------
16      X : array-like
17          The array with imputed values in the target column
18      '''
19      def __init__(self, group_cols, target, metric='mean'):
20
21          assert metric in ['mean', 'median'], 'Unrecognized value for metric, should be mean/medi
22          assert type(group_cols) == list, 'group_cols should be a list of columns'
23          assert type(target) == str, 'target should be a string'
24
25          self.group_cols = group_cols
26          self.target = target
27          self.metric = metric
28
29      def fit(self, X, y=None):
30
31          assert pd.isnull(X[self.group_cols]).any(axis=None) == False, 'There are missing values
32
33          impute_map = X.groupby(self.group_cols)[self.target].agg(self.metric) \
34                                                              .reset_index(drop=False)
35
36          self.impute_map_ = impute_map
37
38          return self
39
40      def transform(self, X, y=None):
41
42          # make sure that the imputer was fitted
43          check_is_fitted(self, 'impute_map_')
44
45          X = X.copy()
46
47          for index, row in self.impute_map_.iterrows():
48              ind = (X[self.group_cols] == row[self.group_cols]).all(axis=1)
49              X.loc[ind, self.target] = X.loc[ind, self.target].fillna(row[self.target])
50
51          return X.values
```

custom_imputer_4.py hosted with ♡ by GitHub view raw

time the custom imputer class is compatible with `scikit-learn`'s `Pipelines`.

So what actually happens in the background? By inheriting from `BaseEstimator` we automatically get the `get_params` and `set_params` methods (all `scikit-learn` estimators require those). Then, inheriting from `TransformerMixin` provides the `fit_transform` method.

> ***Note:*** *There are also other kinds of Mixin classes available for inheritance. Whether we need to do so depends on the type of estimator we want to code. For example,* `ClassifierMixin` *and* `RegressorMixin` *give us access to the* `score` *method used for evaluating the performance of the estimators.*

In the `__init__` method, we stored the input parameters:

- `group_cols` — the list of columns to aggregate over,
- `target` — the target column for imputation (the column in which the missing values are located),
- `metric` — the metric we want to use for imputation, it can be either the mean or the median of the group.

Additionally, we included a set of assertions to make sure we pass in the correct input.

In the `fit` method, we calculate the `impute_map_`, which is a `DataFrame` with the aggregated metric used for imputing. We also check if there are no missing values in the columns we used for aggregation. It is also very important no know that the `fit` method should always return `self`!

Lastly, in the `transform` method we replace the missing values in each group (indicated by the rows of the `impute_map_`) with the appropriate values. As an extra precaution, we use `check_is_fitted` to make sure that we have already fitted the imputer object before using the `transform` method. Before actually transforming the data, we make a copy of it

In both the `fit` and `transform` methods, we have also specified `y=None` in the method definition, even though the `GroupImputer` class will not be using the `y` value of the dataset (also known as the target, not to be confused with the `target` parameter, which indicates the imputation target). The reason for including it is to ensure compatibility with other `scikit-learn` classes.

It is time to see the custom imputer in action!

Running the code prints out the following:

```
df contains 10 missing values.
df_imp contains 0 missing values.
```

As with all imputers in `scikit-learn`, we first create the instance of the object and specify the parameters. Then, we use the `fit_transform` method to create the new object, with the missing values in the `height` column replaced by averages calculated over the `sample_name` and `variant`.

To create `df_imp`, we actually need to manually convert the output of the transformation into a `pd.DataFrame`, as the original output is a `numpy` array. That is the case with all imputers/transformers in `scikit-learn`.

We can see that the imputer worked as expected and replaced all the missing values in our toy `DataFrame`.

Conclusions

In this article, I showed how to quickly create a custom imputer by inheriting from some base classes in `scikit-learn`. This way, the coding is much faster and we also ensure that the imputer is compatible with the entire `scikit-learn` framework.

other projects, as we tried to make it as flexible as possible in the first place.

You can find the code used for this article on my GitHub. As always, any constructive feedback is welcome. You can reach out to me on Twitter or in the comments.

References

[1] https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/impute/_base.py

[2] https://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.base.TransformerMixin.html#sklearn.base.TransformerMixin

Machine Learning Python Data Science Scikit Learn Towards Data Science