Assignment 3

CSCD01 Winter 2021

Spaghetti Code

Julian Barker, Adam Ah-Chong, Andrew Gao, Jesse Francispillai, Laphonso (Poncie) Reyes, Saad Ali

Table of Contents

Introduction 2

Feature #13990 2

Description 2

Design Details 2

Initial State 2

Goal State 3

Bug #17087 5

Description 5

Design Details 5

Initial State: 5

Goal State: 6

Implementation of Feature #13990 7

User Guide 8

sklearn.model_selection.ShuffleSplit 8

sklearn.model_selection.GroupShuffleSplit 9

sklearn.model_selection.StratifiedShuffleSplit 10

sklearn.model_selection.train_test_val_split 11

Customer Acceptance Tests 12

Team Organization 15

Introduction

In a similar fashion to the second assignment, our team began with preparing a list of bugs/features of moderate difficulty or higher to choose from the scikit-learn remote repository, with each team member adding 1 or 2 to the list. Everyone on the team would then read through each issue then vote for the 2 issues that they thought we should do. The 2 issues with the most votes were chosen. Each team member assigned themselves tasks on JIRA, informing the rest of the group what they were working on and any issues or roadblocks they encountered.

Issues selected:
Team decision: https://github.com/scikit-learn/scikit-learn/issues/13990
Team decision: https://github.com/scikit-learn/scikit-learn/issues/14959

Feature #13990

Description

As of right now, the creation of machine learning models in scikit-learn may only consist of including: training data and testing data. Scikit-learn provides the function train_test_split() which splits arrays or matrices into random train and test subsets. As the Github issue suggests, there has been a significant increase of demand to allow splitting 3 data types, rather than 2.The first dataset is the **training dataset**, used for “fitting” the training dataset parameters of the model (using the fit() method). The second dataset is the **validation dataset**, which is used for evaluating how well the model has been trained and to tune parameters of the model. The last dataset is the **test dataset** which is solely used for testing the performance of the model. As previously stated, the current scikit-learn library contains a function train_test_split() to split a dataset into training data and testing data. Therefore, this feature aims to abstract and expand the number of split partitions.

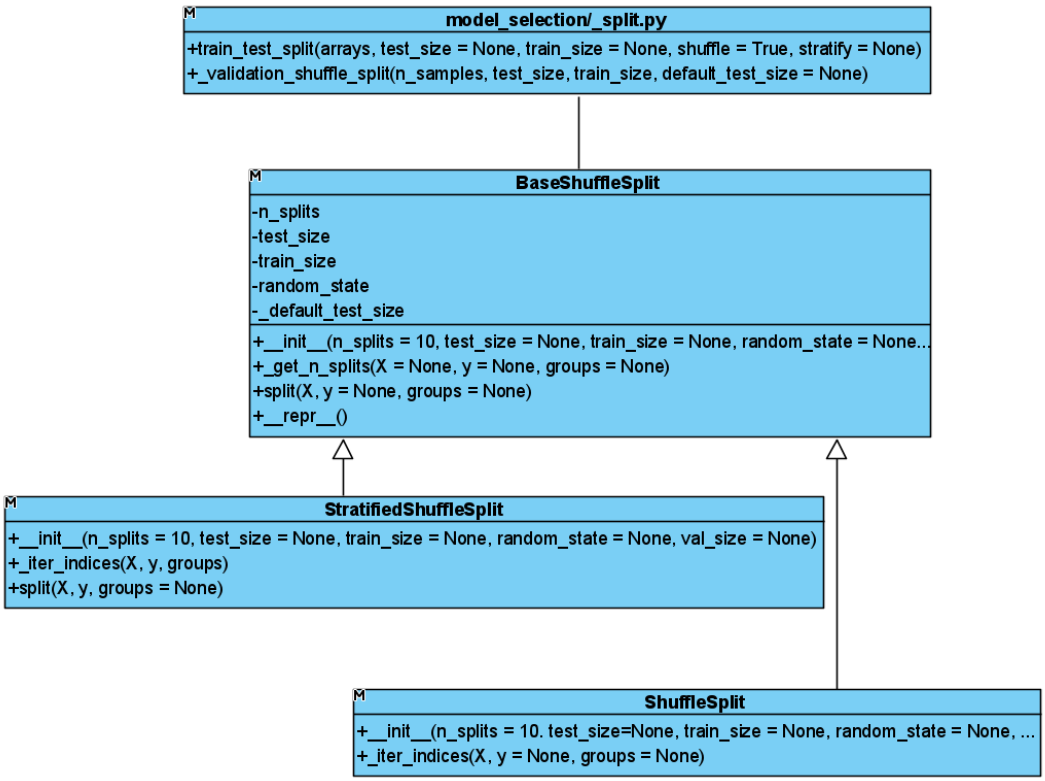
Design Details

In order to implement this feature, the only file that will be modified is **`sklearn/model_selection/_split.py`**.

Initial State

Splitting data into training and testing datasets are handled by a standalone function defined in ***`sklearn/`*****`model_selection/_split.py`** called **`train_test_split()`**. This function is what is used in practice when splitting such datasets. The core functionality relies on a helper function called **`_validate_shuffle_split`**, which in turn deals with validating the imputed datasets and decides how they will be split with respect to a specified proportionality of the dataset sizes. Upon doing so, the function decides which splitting algorithm it will compute based on the shuffle and stratify arguments:

1. If shuffle is set to False
   1.1. If stratify is not None
      - **`ValueError`** occurs as stratified split without shuffling is not implemented.
   1.2. If stratify is None
      - Simply arrange the train and test datasets with respect to what **`_validate_shuffle_split`** outputted.
2. If shuffle is set to True
   2.1. If stratify is set to True
      - The splitting algorithm is defined by the **`StratifiedShuffleSplit`** class.
   2.2. If stratify is set to False
      - The splitting algorithm is defined by the **`ShuffleSplit`** class.

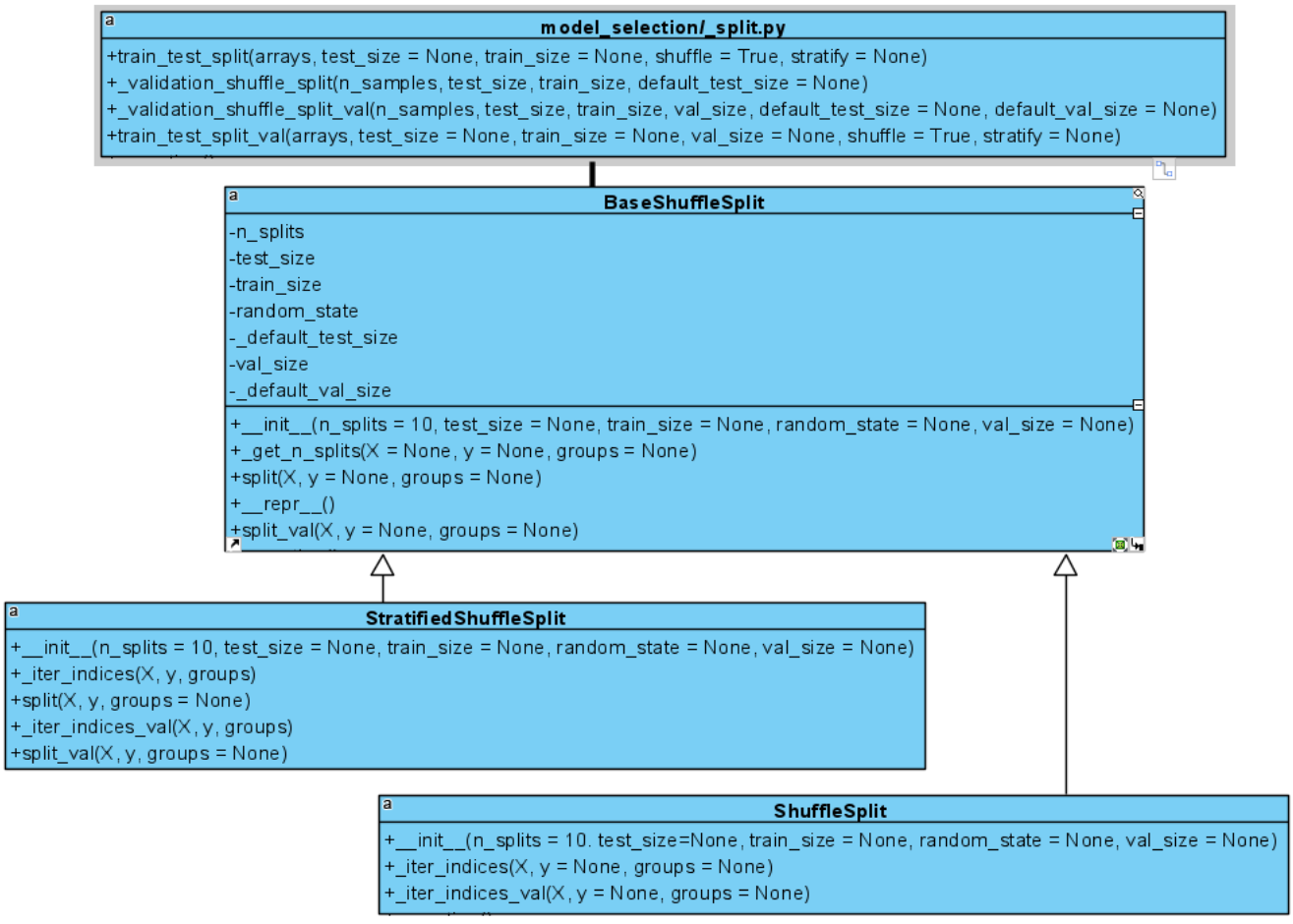


Goal State

In order to implement a function that will split datasets into train, test, and validation subsets, we considered a key factor in our approach: minimize changes and existing functionality to the codebase. Thus, through various design suggestions that our team has internally proposed, we decided to create a new function **train_test_val_split()** which mirrors the functionality of the existing **train_test_split()** function but also includes the validation dataset. This new function will evade from dealing with changes in tests and separate the functionality of the two functions,

as they both serve different purposes. Therefore, the following are the required changes needed:

1. Create a new **train_test_val_split()** function that splits a dataset into train, test, and validation subsets
2. Create a new **_validate_shuffle_val_spllt()** function that deals with sizing the subsets with respect to specified proportions.
3. As previously mentioned, the original **train_test_split()** function decides on a splitting algorithm using the classes **ShuffleSplit** and **StratifiedShuffleSplit**, which both extend their abstract parent class **BaseShuffleSplit**. As a result, the **train_test_val_split()** function will utilize them, but we must extend both of them to incorporate a train,test, and val splitting algorithm.
    a. Hence, we add val_size parameters to each of their constructors and implement **_iter_indices_val** (a new abstract method) which mirrors the functionality of the current **_iter_indices** (current abstract method) but takes validation into account.



Bug #17087

Description

Add ability to feed a ColumnTransformer object to an IterativeImputer instance, along with a list of estimators for each column, and have it return my data in its original format with imputed values.

Design Details

The current state of the IterativeImputer class within scikit-learn uses a single Estimator instance to predict data and fill in missing values. This Estimator can be of any type.

For this feature, we intend to add functionality that allows us to specify use multiple Estimators for predicting datapoints in different columns. This can be useful for datasets that contain many different components for each data point (e.g. the Titanic data which has both Integer-type and String-type values for each data points).

We also wish to be able to pick which kind of Transformer the IterativeImputer uses (as currently we can only use a default Transformer, as this is hard-coded into the IterativeImputer initialization). In this case, the IterativeImputer also uses an instance of a single Transformer that we can specify..

While we would like for any Transformer to be compatible with IterativeImputer, for this feature we are focusing on compatibility with a ColumnTransformer, as ColumnTransformer would be a very useful Transformer to use with our new multi-column Estimator functionality.

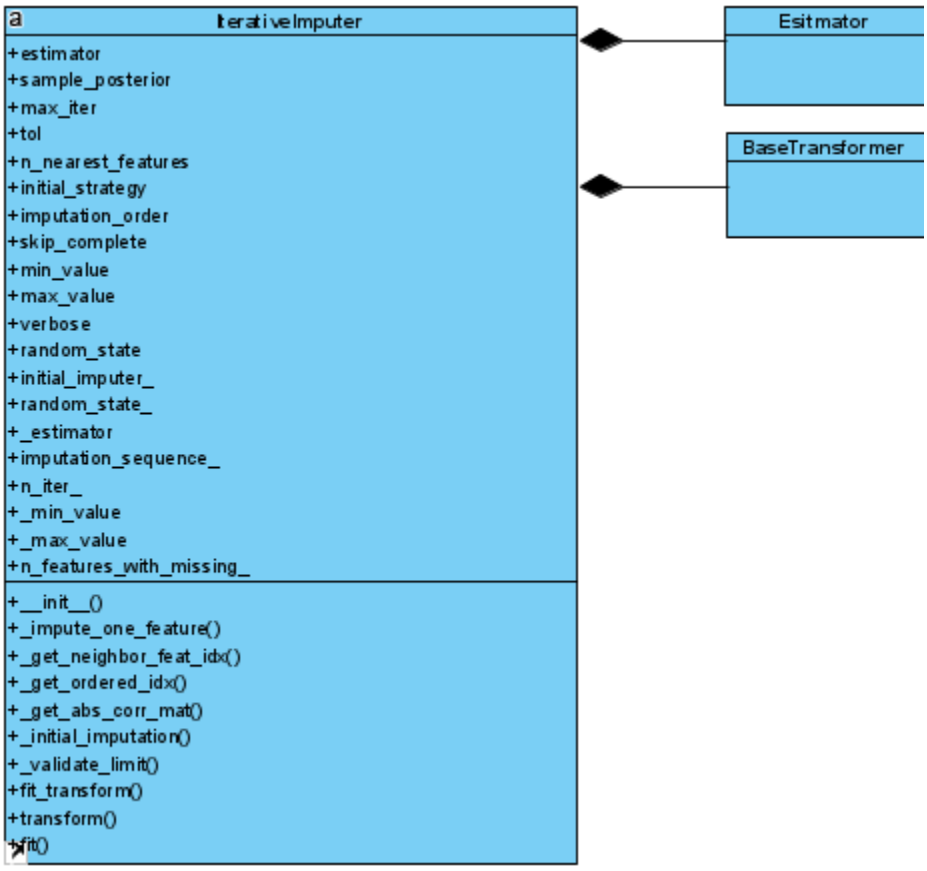
Initial State:

The IterativeImputer class is an Estimator which is used to fill in missing data points within datasets. This is a very useful feature, as real life is messy, and recorded data is likely to have discrepancies because of it.

The initial state of the IterativeImputer class interacts with Estimator and Transformer classes in a rather rigid way. Currently, IterativeImputer instances can only use a single Estimator type for its imputation process. This Estimator type is also used for the imputation process across all fields available within the dataset. This means that the values of multiple fields with very different kinds of data may still influence each other as the IterativeImputer uses one field to estimate the possible value of another.

Another source of inflexibility is the use of Transformers within IterativeImputer. IterativeImputer instances only use basic Transformer functionality in order to make sure that inputted data is of the correct dimensions. If the user wishes to transform the data in any way, they are not able to specify what type of Transformer they wish to use with the IterativeImputer, meaning they would

have to either transform the data before inputting it into the IterativeImputer, or find a workaround if pre-transforming is not viable within their work pipeline.



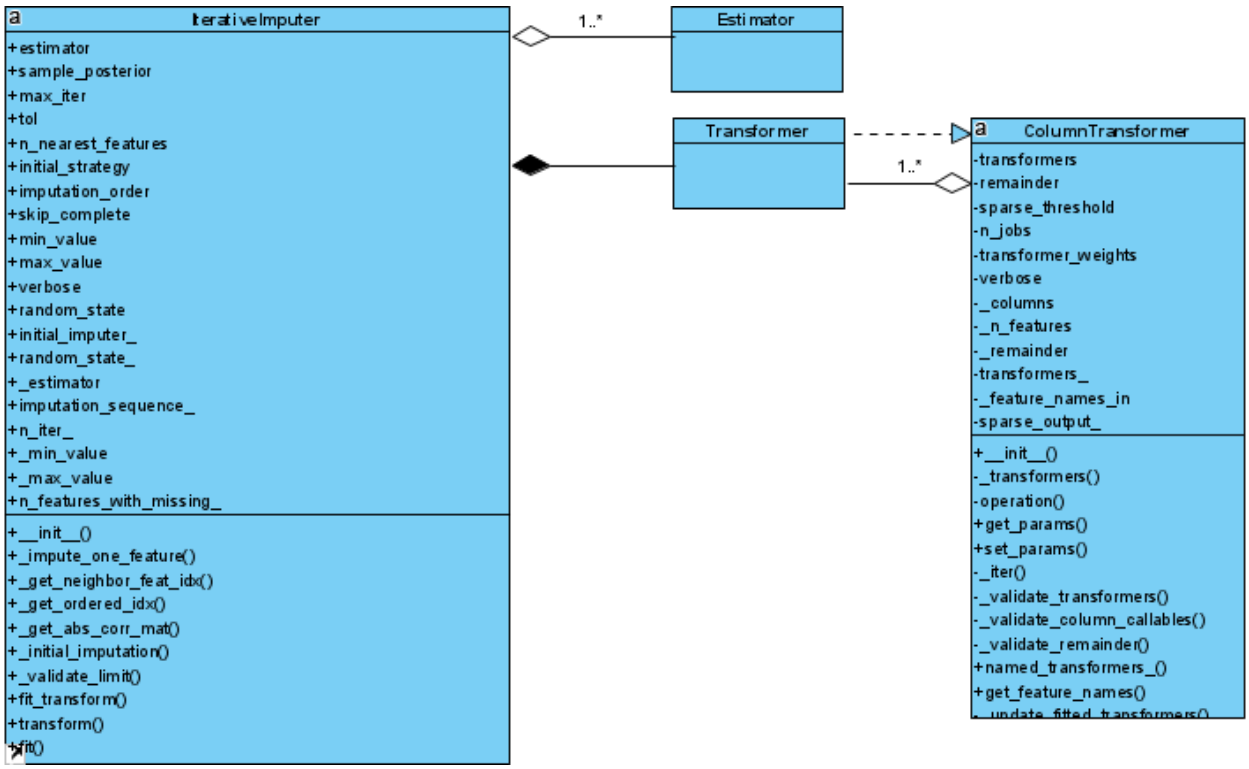
Goal State:

The goal of this feature is two allow the IterativeImputer to transform and impute data across multiple differing types of fields, specifically fields of different variable types. This would allow for cleaner imputation across datasets without having to delve deep into making custom pipelines for each type of multivariate dataset a user may want to impute.

In order to create a working version of this feature, the following changes will need to be made:

- IterativeImputer must be able to take in a Transformer type as part of its input or instantiation, and be able to use the specified Transformer to transform inputted data.
    - We are focusing especially on compatibility with inputting the ColumnTransformer type, as it would allow for functionality like transforming fields of one variable type into another for homogenous data types and cleaner imputing processes
- IterativeImputer must be able to take in a number Estimator types as part of its input or instantiation. The user should also be able to specify which fields each Estimator type will affect, as this will allow the user to use Estimators that are appropriate for each unique field in the dataset. Functionality needs to be added that makes it so

IterativeImputer can impute values from different fields using only the Estimator specified for that specific field.



Implementation of Feature #13990

As a team, we have decided to implement feature #13990. This issue has been in high demand from various users, as generating train, testing, and validation datasets has practical use. For instance, here is a link to stackoverflow listing posts related to scikit-learn and the common trend of users demanding for splitting data into train,test, and validation: https://stackoverflow.com/questions/tagged/scikit-learn?sort=frequent&pageSize=50.
We found that the design and implementation process for this issue was deemed more interesting and useful to the community. Furthermore, the complexity of it was labeled as “Moderate”, however, it entailed deep research in understanding mathematical constructs used in the current train and test splitting function, and how these computations differ in train, test, and validation splitting. As a result, the experience we gained completing this issue was more valuable for us than we initially expected.

The second feature we looked at, issue #17087, seemed like a straightforward and interesting feature to implement at first. Adding multivariate compatibility to an estimator that already works with multidimensional arrays definitely seemed like it would be possible with the current tools that scikit-learn provides. While we initially planned to integrate the functionality of Transformer classes with the IterativeImputer class we were focusing on, looking closer at the code revealed to us that the scope of this task was actually much larger than it initially seemed. Several design

decisions made a year or two ago for scikit-learn's development proved to cut off a lot of the functionality and flexibility that would have made this task much more viable. Upon realizing that we would not be able to implement the requested feature without restructuring currently existing Transformer and Estimator classes, or rewriting them with modifications that suited our needs, both of which were beyond the limits of our timeframe, we decided on implementing feature #13990, to avoid falling into having to develop beyond our scope and have a viable product ready by the deadline provided.

User Guide

sklearn.model_selection.ShuffleSplit

(Parts of the user guide that already existed and were not changed are omitted from this section. The parts that were changed or added are underlined.)

class `sklearn.model_selection.`**`ShuffleSplit`**(*n_splits=10*, *, val_size=None, *test_size=None*, *train_size=None*, *random_state=None*)

Random permutation cross-validator

Yields indices to split data into training, test and validation sets.

Note: contrary to other cross-validation strategies, random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Parameters

val_size : *float* or *int*, default=None
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the validation split. If int, represents the absolute number of validation samples. If None, the value is automatically set to the complement of the test and train size.

Methods

`split_val`(*X*, *y=None*, *groups=None*)

Generate indices to split data into train, test and validation sets.

Parameters:

X : array-like of shape (n_samples, n_features)
Training data, where n_samples is the number of samples and n_features is the number of features.

y : array-like of shape (n_samples,)
The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
Group labels for the samples used while splitting the dataset into train/test set.

Returns:

train : ndarray
The training set indices for that split.

test : ndarray
The testing set indices for that split.

val: ndarray
The validation set indices for that split.

sklearn.model_selection.GroupShuffleSplit

(Parts of the user guide that already existed and were not changed are omitted from this section. The parts that were changed or added are underlined.)

class `sklearn.model_selection.`**`GroupShuffleSplit`**(*n_splits=5*, ***, *val_size=None, test_size=None*, *train_size=None*, *random_state=None*)

Parameters

val_size : *float* or *int*, default=None
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the validation split. If int, represents the absolute number of validation samples. If None, the value is automatically set to the complement of the test and train size.

Methods

`split_val`(*X*, *y=None*, *groups=None*)

Generate indices to split data into train, test and validation sets.

Parameters:

X : array-like of shape (n_samples, n_features)
Training data, where n_samples is the number of samples and n_features is the number of features.

y : array-like of shape (n_samples,)
The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
Group labels for the samples used while splitting the dataset into train/test set.

Returns:

train : ndarray
The training set indices for that split.

test : ndarray
The testing set indices for that split.

val: ndarray
The validation set indices for that split.

sklearn.model_selection.StratifiedShuffleSplit

(Parts of the user guide that already existed and were not changed are omitted from this section. The parts that were changed or added are underlined.)

class `sklearn.model_selection.`**`StratifiedShuffleSplit`**(*n_splits=10*, ***, val_size=None, *test_size=None*, *train_size=None*, *random_state=None*)

Stratified ShuffleSplit cross-validator

Provides train/test indices to split data in train/test/validation sets.

This cross-validation object is a merge of StratifiedKFold and ShuffleSplit, which returns stratified randomized folds. The folds are made by preserving the percentage of samples for each class.

Note: like the ShuffleSplit strategy, stratified random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Parameters

val_size : *float* or *int*, default=None
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the validation split. If int, represents the absolute number of validation samples. If None, the value is automatically set to the complement of the test and train size.

Methods

`split_val`(*X*, *y=None*, *groups=None*)

Generate indices to split data into train, test and validation sets.

Parameters:

X : array-like of shape (n_samples, n_features)
Training data, where n_samples is the number of samples and n_features is the number of features.

y : array-like of shape (n_samples,)
The target variable for supervised learning problems.

groups : array-like of shape (n_samples,), default=None
Group labels for the samples used while splitting the dataset into train/test set.

Returns:

train : ndarray
The training set indices for that split.

test : ndarray
The testing set indices for that split.

val: ndarray

The validation set indices for that split.

sklearn.model_selection.train_test_val_split

(This section is new and everything in it is a new addition to the user guide)

sklearn.model_selection.train_test_val_split(*arrays, val_size=None, test_size=None, train_size=None, random_state=None, shuffle=True, stratify=None)

Split arrays or matrices into random train, test, and validation subsets

Quick utility that wraps input validation and next(ShuffleSplit().split(X, y)) and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the User Guide.

Parameters

arrays : *sequence of indexables with same length / shape[0]
Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.
val_size : *float* or *int*, *default=None*
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the val split. If int, represents the absolute number of val samples. If None, the value is set to the complement of the train size + test size. If `test_size` is also None, it will be set to 0.2. If `train_size` is also None, it will be set to 0.2, for a 60%, 20%, 20% split.
test_size : *float or int*, *default=None*
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If train_size is also None, it will be set to 0.25.
train_size : *float or int*, *default=None*
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.
random_state : *int, RandomState instance or None, default=None*
Controls the shuffling applied to the data before applying the split. Pass an int for reproducible output across multiple function calls. See Glossary.
shuffle : *bool, default=True*
Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.
stratify : *array-like, default=None*
If not None, data is split in a stratified fashion, using this as the class labels. Read more in the User Guide.

Returns:

splitting : ***list, length=2 * len(arrays)***

List containing train-test-val split of inputs.

Customer Acceptance Tests

Train_test_val_split allows us to do a threefold split which was previously not possible to do. To try the new feature we can take the following steps.

1. Import the train_test_val_split

```
from sklearn.model_selection import train_test_val_split
```

2. Setup up the X value to train the data on

3. Next we call train test_val_split with all the values being None

X_train, X_test, X_val = train_test_val_split(X, train_size=None,test_size=None, val_size=None)

```
X = [[4,5],[8,7],[2,5],[9,6],[5,4],[1,2],[5,9],[8,7],[6,5],[4,3]]

X_train, X_test, X_val = train_test_val_split(X, train_size=None,test_size=None, val_size=None)
```

4. When we check the value of X_train, X_test, X_val we should get the proper size as specified by the train_size, test_size and val_size parameters.

Values of the variables will not be the same every time however the size should remain the same as specified from the variables.

X_train = [[8, 7], [9, 6], [4, 3], [1, 2], [6, 5], [5, 9]] should look something like this with len == 6

X_test = [[2, 5], [8, 7]] should look something like this with len == 2

X_val = [[5, 4], [4, 5]] should look something like this with len == 2

With the values of the variables being set to None the default split is 60% will be in train and 20% in X_test and 20% in X_val as wanted. We can change the split to however we want as outlined in the next step

5. Now let's call the same function with train_size be None, test_size be 1 and val_size be 5

We should get 4 elements in X_train, 1 in test and, 5 in val_size

```
X_train, X_test, X_val = train_test_val_split(X, train_size=None,test_size

=1, val_size=5)
```

6. Train_test_val_split is also able to take floats as parameters for example 4 is the same as 0.4 so if we where to do the same test above this time changing test_size to 0.1 the same outcome should occur

7. Now try setting one of the parameters to a negative number, a value error should be thrown as the range must either be a float between 0 and 1 or a positive integer.

8. Now lets try with an X and an y value first we will import numpy as well as import coo_matrix

```
import numpy as np

from scipy.sparse import coo_matrix
```

9. Now setup up the variable as following, here we are creating a matrix to train data on

```
X = np.arange(100).reshape((10,10))

X_s = coo_matrix(X)

y = np.arange(10)
```

10. Call train_test_val_split with the X and y value.

```
split=train_test_val_split(X, y, test_size=None, train_size=None, val_size=None)

X_train, X_test, X_val, y_train, y_test, y_val = split
```

Taking a look at the value of X_train, X_test, and X_val we will notice that 60% of the data is in X_train, 40% in X_test and X_val as the size values were all set to None.

When looking at the values of X_train and y_train we should notice that the 0 indexes of the X_train values should be 10 times the value of the y train value. This is true for test and val for both X and y as well.

We can change around the value of the size parameters to change the size of the training data and the test size which will be reflected in the output.

Team Organization

Assignment Goal

- **Design and plan the implementation and testing of a new feature, or a significant bug fix, for scikit-learn**.
- *Selecting a feature that can be fully implemented, documented, and tested in the time available.*

Meeting 1: 2021/03/09

The idea for this meeting is to discuss what needs to be done and how we will approach completing this by Meeting 2

1. From the issue list, select **at least two features/bug fixes** to examine further.
    a. **MEDIUM** difficulty
2. Briefly and clearly describe the selected features/bug fixes in your report.

Meeting 2: 2021/03/12

1. For each of the selected features/bug fixes, investigate what parts of the existing code base you would need to modify in order to implement the feature. Produce designs for the selected feature, describing your plans for the organization of new code, as well as all interactions between new code and existing code. You guessed it: UML diagrams would be very helpful here.
2. Select one of these features/bug fixes for implementation, and briefly explain your decision.
3. Update JIRA

Team decision: https://github.com/scikit-learn/scikit-learn/issues/13990
Team decision: https://github.com/scikit-learn/scikit-learn/issues/14959

Meeting 3: 2021/03/21

1. For the selected feature/bug fix, produce a suite of acceptance tests that will demonstrate that the feature/bug fix has been implemented correctly. Design these as ”customer acceptance” tests: i.e. a description of the steps a user needs to carry out to check that the program works as expected.
2. For the selected feature/bug fix produce a unit test suite that will demonstrate that the feature/bug fix has been implemented correctly.
3. Update JIRA