

---

# **pyDataset**

***Release 0.16.3***

**J. Renero**

**Jul 11, 2021**



## **A COMPREHENSIVE TUTORIAL WITH EXAMPLES:**

|          |                              |           |
|----------|------------------------------|-----------|
| <b>1</b> | <b>Install</b>               | <b>3</b>  |
| <b>2</b> | <b>Data Tutorial / Guide</b> | <b>5</b>  |
| <b>3</b> | <b>The API Documentation</b> | <b>27</b> |
| <b>4</b> | <b>Indices and tables</b>    | <b>41</b> |
|          | <b>Python Module Index</b>   | <b>43</b> |
|          | <b>Index</b>                 | <b>45</b> |



J. Renero

Dataset is for educational purposes, mainly. It tries to help those approaching Data Science in Python for the first time, who must deal with common (and time consuming) data preparation tasks.

This package tries, through a very simple approach, to collect all the common tasks that are normally done over pandas DataFrames, like:

- load data
- set the target variable
- describe the health status of the dataset
- drop/keep columns or sample from simple lists
- split the dataset
- count categorical and numerical features
- fix NA's
- find correlations
- detect skewness
- scale numeric values
- detect outliers
- one hot encoding
- find under represented categorical features
- perform stepwise feature selection
- compute information gain,
- plot some useful charts



## INSTALL

To install this package, simply *pip* from this git repo:

```
$ pip install git+https://github.com/renero/dataset
```





## DATA TUTORIAL / GUIDE

### Table of Contents

#### 1 Dataset introduction

##### 1.1 Data loading

###### 1.1.1 Access to internal DataFrame

###### 1.1.2 Access to numerical/categorical variables

###### 1.1.3 Set the target variable

###### 1.1.4 Access to features

##### 1.2 Data description

##### 1.3 Data Manipulation

###### 1.3.1 Remove columns

###### 1.3.2 Add columns

###### 1.3.3 Remove samples

###### 1.3.4 Samples Matching criteria

##### 1.4 Data Conversion

###### 1.4.1 Type conversion (int, float)

###### 1.4.2 Categorical <-> Numerical

#### 2 Histograms and density plots

##### 2.1 Histograms

##### 2.2 Density plots

##### 2.3 Features importance

##### 2.4 Covariance Matrix

##### 2.5 Native pandas plots

#### 3 Data cleaning

##### 3.1 NA's

###### 3.1.1 Replace NA

##### 3.2 Outliers

##### 3.3 Correlation

### 3.4 Under represented features

#### 3.4.1 Merge categories

#### 3.4.2 Merge values

### 4 Data transformation

#### 4.1 One-hot encoding

#### 4.2 Discretize

#### 4.3 Skewness

#### 4.4 Scale

### 5 Feature selection

#### 5.1 Information Gain

#### 5.2 Stepwise feature selection

### 6 Dataset Split

```
[3]: from dataset import Dataset
import numpy as np
import matplotlib.pyplot as plt
```

## 2.1 Dataset introduction

The idea with Dataset is that you simplify most of the tasks that you normally do with pandas DataFrame. This normally applies when you're starting in Python. You will have access at any time, to the underlying pandas DataFrame that holds the data, in case you need to use the numpy representation of the values, or access specific locations of your data.

### 2.1.1 Data loading

To start with Dataset, you must load your data the same way it is done with pandas, by passing the URL or file location to the constructor (Dataset()). If you need to add more pandas parameters to this call, specifying what is the separator, or whether to use headers, etc., simply add them after the file location.

I'm using the location of the CSV from U.Arizona.

```
[4]: URL="https://www2.cs.arizona.edu/classes/cs120/fall17/ASSIGNMENTS/assg02/Pokemon.csv"
pokemon = Dataset(URL, delimiter=',', header=0)
```

From this point, you have access to the methods provided by Dataset to describe the dataset, clean up your data, perform feature selection or plot some interesting feature engineering related plots.

If you already have a DataFrame and want to use inside a Dataset class, you can also import it, using the method:

```
>>> my_dataset = Dataset.from_dataframe(my_dataframe)
```

## Access to internal DataFrame

If you want access your DataFrame you simply have to call the property features at the end of the Dataset name:

```
[5]: pokemon.features
```

```
[5]:
```

|     | #     | Name                  | Type 1  | Type 2 | Total | HP   | Attack | \ |
|-----|-------|-----------------------|---------|--------|-------|------|--------|---|
| 0   | 1.0   | Bulbasaur             | Grass   | Poison | 318.0 | 45.0 | 49.0   |   |
| 1   | 2.0   | Ivysaur               | Grass   | Poison | 405.0 | 60.0 | 62.0   |   |
| 2   | 3.0   | Venusaur              | Grass   | Poison | 525.0 | 80.0 | 82.0   |   |
| 3   | 3.0   | VenusaurMega Venusaur | Grass   | Poison | 625.0 | 80.0 | 100.0  |   |
| 4   | 4.0   | Charmander            | Fire    | NaN    | 309.0 | 39.0 | 52.0   |   |
| ..  | ...   | ...                   | ...     | ...    | ...   | ...  | ...    |   |
| 795 | 719.0 | Diancie               | Rock    | Fairy  | 600.0 | 50.0 | 100.0  |   |
| 796 | 719.0 | DiancieMega Diancie   | Rock    | Fairy  | 700.0 | 50.0 | 160.0  |   |
| 797 | 720.0 | HoopaaHoopaa Confined | Psychic | Ghost  | 600.0 | 80.0 | 110.0  |   |
| 798 | 720.0 | HoopaaHoopaa Unbound  | Psychic | Dark   | 680.0 | 80.0 | 160.0  |   |
| 799 | 721.0 | Volcanion             | Fire    | Water  | 600.0 | 80.0 | 110.0  |   |

|     | Defense | Sp. Atk | Sp. Def | Speed | Generation | Legendary |
|-----|---------|---------|---------|-------|------------|-----------|
| 0   | 49.0    | 65.0    | 65.0    | 45.0  | 1.0        | False     |
| 1   | 63.0    | 80.0    | 80.0    | 60.0  | 1.0        | False     |
| 2   | 83.0    | 100.0   | 100.0   | 80.0  | 1.0        | False     |
| 3   | 123.0   | 122.0   | 120.0   | 80.0  | 1.0        | False     |
| 4   | 43.0    | 60.0    | 50.0    | 65.0  | 1.0        | False     |
| ..  | ...     | ...     | ...     | ...   | ...        | ...       |
| 795 | 150.0   | 100.0   | 150.0   | 50.0  | 6.0        | True      |
| 796 | 110.0   | 160.0   | 110.0   | 110.0 | 6.0        | True      |
| 797 | 60.0    | 150.0   | 130.0   | 70.0  | 6.0        | True      |
| 798 | 60.0    | 170.0   | 130.0   | 80.0  | 6.0        | True      |
| 799 | 120.0   | 130.0   | 90.0    | 70.0  | 6.0        | True      |

[800 rows x 13 columns]

If, instead of displaying the entire pandas DataFrame we want to see a special feature, we can refer to that feature using its name, right after the features property. In this case, let's have a look to the feature called Name holding the pokemon name.

```
[6]: pokemon.features.Name
```

```
[6]:
```

|     |                       |
|-----|-----------------------|
| 0   | Bulbasaur             |
| 1   | Ivysaur               |
| 2   | Venusaur              |
| 3   | VenusaurMega Venusaur |
| 4   | Charmander            |
| ..  | ...                   |
| 795 | Diancie               |
| 796 | DiancieMega Diancie   |
| 797 | HoopaaHoopaa Confined |
| 798 | HoopaaHoopaa Unbound  |
| 799 | Volcanion             |

Name: Name, Length: 800, dtype: object

The result from this call is pandas Series.

or (to show only the first 5 lines from the dataframe):

```
[7]: pokemon.features.head(3)
```

```
[7]:      #      Name Type 1  Type 2  Total    HP  Attack  Defense  Sp. Atk  \
0  1.0  Bulbasaur  Grass  Poison  318.0  45.0   49.0    49.0    65.0
1  2.0   Ivysaur  Grass  Poison  405.0  60.0   62.0    63.0    80.0
2  3.0  Venusaur  Grass  Poison  525.0  80.0   82.0    83.0   100.0

      Sp. Def  Speed  Generation  Legendary
0      65.0   45.0           1.0      False
1      80.0   60.0           1.0      False
2     100.0   80.0           1.0      False
```

Number of features and number of samples in the dataset are accessible through the following properties:

```
[8]: print('Nr of features:', pokemon.num_features)
     print('Nr of samples:', pokemon.num_samples)
```

```
Nr of features: 13
Nr of samples: 800
```

### Access to numerical/categorical variables

It is also possible to work only with the numerical or categorical variables in the dataset. To do that you just have to use the properties: `.categorical` or `.numerical` to access those portions of the dataframe that only contain those feature subtypes.

If we want to access only the categorical, we type:

```
[9]: pokemon.categorical.head(3)
```

```
[9]:      Name Type 1  Type 2  Legendary
0  Bulbasaur  Grass  Poison      False
1   Ivysaur  Grass  Poison      False
2   Venusaur  Grass  Poison      False
```

If we want to access the numerical ones:

```
[10]: pokemon.numerical.head(3)
```

```
[10]:      #  Total    HP  Attack  Defense  Sp. Atk  Sp. Def  Speed  Generation
0  1.0  318.0  45.0   49.0    49.0    65.0    65.0   45.0           1.0
1  2.0  405.0  60.0   62.0    63.0    80.0    80.0   60.0           1.0
2  3.0  525.0  80.0   82.0    83.0   100.0   100.0   80.0           1.0
```

In case we want only the names of the variables that are numerical or categorical, we can either use:

```
[11]: pokemon.categorical_features
```

```
[11]: ['Name', 'Type 1', 'Type 2', 'Legendary']
```

or

```
[12]: pokemon.names('categorical')
```

```
[12]: ['Name', 'Type 1', 'Type 2', 'Legendary']
```

which, of course, also applies to numerical.

## Set the target variable

At this point, we can make something very interesting when working with datasets, which is to select what feature will be the target variable. By doing so, Dataset will separate that feature from the rest, allowing to use special feature engineering methods that we will see afterwards:

```
[13]: pokemon.set_target('Legendary');
```

When we select the target, that feature disappears from the `features` property, as you can see when we call the `head` method again:

```
[14]: pokemon.features.head(3)
```

```
[14]:
```

|   | #   | Name      | Type 1 | Type 2 | Total | HP   | Attack | Defense | Sp. Atk | \ |
|---|-----|-----------|--------|--------|-------|------|--------|---------|---------|---|
| 0 | 1.0 | Bulbasaur | Grass  | Poison | 318.0 | 45.0 | 49.0   | 49.0    | 65.0    |   |
| 1 | 2.0 | Ivysaur   | Grass  | Poison | 405.0 | 60.0 | 62.0   | 63.0    | 80.0    |   |
| 2 | 3.0 | Venusaur  | Grass  | Poison | 525.0 | 80.0 | 82.0   | 83.0    | 100.0   |   |

|   | Sp. Def | Speed | Generation |
|---|---------|-------|------------|
| 0 | 65.0    | 45.0  | 1.0        |
| 1 | 80.0    | 60.0  | 1.0        |
| 2 | 100.0   | 80.0  | 1.0        |

Our feature is now in a new property called `target`. We can access it, by calling it from our Dataset, which will return a pandas Series object to work with.

```
[15]: pokemon.target
```

```
[15]:
```

|     |       |
|-----|-------|
| 0   | False |
| 1   | False |
| 2   | False |
| 3   | False |
| 4   | False |
| ... |       |
| 795 | True  |
| 796 | True  |
| 797 | True  |
| 798 | True  |
| 799 | True  |

Name: Legendary, Length: 800, dtype: bool

If at any point during your work you want to unset the target variable, and make it part of the dataset again as a normal feature, you just have to call

```
>>> pokemon.unset_target()
```

From that point, no target variable is defined within the dataset and all features are considered normal features.

## Access to features

From this point, if we want to access a DataFrame that will contain all features, including the target variable, we must use the property `all`, because, as you can see, `features` no longer contains the target variable:

```
[16]: pokemon.all.head(3)
```

```
[16]:
```

|   | #   | Name      | Type 1 | Type 2 | Total | HP   | Attack | Defense | Sp. Atk | \ |
|---|-----|-----------|--------|--------|-------|------|--------|---------|---------|---|
| 0 | 1.0 | Bulbasaur | Grass  | Poison | 318.0 | 45.0 | 49.0   | 49.0    | 65.0    |   |
| 1 | 2.0 | Ivysaur   | Grass  | Poison | 405.0 | 60.0 | 62.0   | 63.0    | 80.0    |   |
| 2 | 3.0 | Venusaur  | Grass  | Poison | 525.0 | 80.0 | 82.0   | 83.0    | 100.0   |   |

|   | Sp. Def | Speed | Generation | Legendary |
|---|---------|-------|------------|-----------|
| 0 | 65.0    | 45.0  | 1.0        | False     |
| 1 | 80.0    | 60.0  | 1.0        | False     |
| 2 | 100.0   | 80.0  | 1.0        | False     |

## 2.1.2 Data description

First thing we can do with our dataset is to describe it, just to know the types of the variables, and whether we have NA's or incomplete features.

```
[17]: pokemon.describe()
```

```
12 Features. 800 Samples
Available types: [dtype('float64') dtype('O')]
  · 3 categorical features
  · 9 numerical features
  · 1 categorical features with NAs
  · 0 numerical features with NAs
  · 12 Complete features
--
Target: Legendary (bool)
'Legendary'
  · Min.: 0.0000
  · 1stQ: 0.0000
  · Med.: 0.0000
  · Mean: 0.0813
  · 3rdQ: 0.0000
  · Max.: 1.0000
```

In case you want more information about the values, each feature is taking, then you can use the `summary()` method:

```
[18]: pokemon.summary()
```

```
Features Summary (all):
'#'          : float64    Min.(1.0) 1stQ(184.) Med.(364.) Mean(362.) 3rdQ(539.) Max.(721.)
'Name'       : object      800 categs. 'Bulbasaur'(1, 0.0013) 'Ivysaur'(1, 0.0013)
↳ 'Venusaur'(1, 0.0013) 'VenusaurMega Venusaur'(1, 0.0013) ...
'Type 1'     : object      18 categs. 'Grass'(112, 0.1400) 'Fire'(98, 0.1225) 'Water'(70,
↳ 0.0875) 'Bug'(69, 0.0862) ...
'Type 2'     : object      18 categs. 'Poison'(97, 0.2343) 'nan'(35, 0.0845) 'Flying'(34,
↳ 0.0821) 'Dragon'(33, 0.0797) ...
'Total'      : float64    Min.(180.) 1stQ(330.) Med.(450.) Mean(435.) 3rdQ(515.) Max.(780.
↳ )
```

(continues on next page)

(continued from previous page)

```

'HP'      : float64    Min.(1.0) 1stQ(50.0) Med.(65.0) Mean(69.2) 3rdQ(80.0) Max.(255.)
'Attack'   : float64    Min.(5.0) 1stQ(55.0) Med.(75.0) Mean(79.0) 3rdQ(100.) Max.(190.)
'Defense'  : float64    Min.(5.0) 1stQ(50.0) Med.(70.0) Mean(73.8) 3rdQ(90.0) Max.(230.)
'Sp. Atk'  : float64    Min.(10.0) 1stQ(49.7) Med.(65.0) Mean(72.8) 3rdQ(95.0) Max.(194.
↪)
'Sp. Def'  : float64    Min.(20.0) 1stQ(50.0) Med.(70.0) Mean(71.9) 3rdQ(90.0) Max.(230.
↪)
'Speed'    : float64    Min.(5.0) 1stQ(45.0) Med.(65.0) Mean(68.2) 3rdQ(90.0) Max.(180.)
'Generation': float64    Min.(1.0) 1stQ(2.0) Med.(3.0) Mean(3.32) 3rdQ(5.0) Max.(6.0)
'Legendary': bool       2 categs. 'False'(735, 0.9187) 'True'(65, 0.0813)

```

## 2.1.3 Data Manipulation

### Remove columns

You can easily remove columns from data by using the method `drop_columns()`. You can pass a single column/feature name or a list of features, and they will disappear from the dataset.

```

[19]: pokemon.drop_columns('#')
[19]: <dataset.dataset.Dataset at 0x133a668d0>

```

You can also remove all the columns that are not in a list. To do that, you use `keep_columns()` and what you must pass to the function is a feature name or a list of feature names you want to keep in your dataset. For example, if we might want to keep only the numerical features:

```
pokemon.keep_columns(pokemon.numerical_features)
```

or if we might want to keep only a couple of well known features:

```
pokemon.keep_columns(['Total', 'Attack'])
```

### Add columns

You can also add columns or entire dataframes to your existing dataset. If you want to simply add a pandas Series to the existing dataset, call:

```
pokemon.add_columns(my_data_series)
```

If what you want is to add an entire dataframe, the mechanism is exactly the same:

```
pokemon.add_columns(my_dataframe)
```

In the following cells, all features are shown, then categoricals are extracted to variable `categoricals`, and removed from the dataset. Then `categoricals` is added to the original dataset, resulting in a dataset which is equivalent to the one we started with.

```

[20]: # Original dataset
pokemon.features.head(3)

```

```
[20]:
```

|   | Name      | Type 1 | Type 2 | Total | HP   | Attack | Defense | Sp. Atk | Sp. Def | \ |
|---|-----------|--------|--------|-------|------|--------|---------|---------|---------|---|
| 0 | Bulbasaur | Grass  | Poison | 318.0 | 45.0 | 49.0   | 49.0    | 65.0    | 65.0    |   |
| 1 | Ivysaur   | Grass  | Poison | 405.0 | 60.0 | 62.0   | 63.0    | 80.0    | 80.0    |   |
| 2 | Venusaur  | Grass  | Poison | 525.0 | 80.0 | 82.0   | 83.0    | 100.0   | 100.0   |   |

|   | Speed | Generation |
|---|-------|------------|
| 0 | 45.0  | 1.0        |
| 1 | 60.0  | 1.0        |
| 2 | 80.0  | 1.0        |

```
[21]: categoricals = pokemon.categorical

pokemon.drop_columns(pokemon.categorical_features)
pokemon.features.head(3)
```

```
[21]:
```

|   | Total | HP   | Attack | Defense | Sp. Atk | Sp. Def | Speed | Generation |
|---|-------|------|--------|---------|---------|---------|-------|------------|
| 0 | 318.0 | 45.0 | 49.0   | 49.0    | 65.0    | 65.0    | 45.0  | 1.0        |
| 1 | 405.0 | 60.0 | 62.0   | 63.0    | 80.0    | 80.0    | 60.0  | 1.0        |
| 2 | 525.0 | 80.0 | 82.0   | 83.0    | 100.0   | 100.0   | 80.0  | 1.0        |

```
[22]: pokemon.add_columns(categoricals)
pokemon.features.head(3)
```

```
[22]:
```

|   | Total | HP   | Attack | Defense | Sp. Atk | Sp. Def | Speed | Generation | \ |
|---|-------|------|--------|---------|---------|---------|-------|------------|---|
| 0 | 318.0 | 45.0 | 49.0   | 49.0    | 65.0    | 65.0    | 45.0  | 1.0        |   |
| 1 | 405.0 | 60.0 | 62.0   | 63.0    | 80.0    | 80.0    | 60.0  | 1.0        |   |
| 2 | 525.0 | 80.0 | 82.0   | 83.0    | 100.0   | 100.0   | 80.0  | 1.0        |   |

|   | Name      | Type 1 | Type 2 |
|---|-----------|--------|--------|
| 0 | Bulbasaur | Grass  | Poison |
| 1 | Ivysaur   | Grass  | Poison |
| 2 | Venusaur  | Grass  | Poison |

## Remove samples

If what you want is to remove some samples (rows) from the dataset, you simply call the method `drop_samples()` passing the list of indices you want to remove. For example:

```
pokemon.drop_samples([34, 56, 78])
```

## Samples Matching criteria

If you want to select samples for which one of the features fulfills a certain criteria, you can get the list of indices of those samples by calling:

```
[23]: print('There are', len(pokemon.samples_matching('Grass', 'Type 1')),
        'samples for which \'Type 1\' is valued \'Grass\'')

There are 70 samples for which 'Type 1' is valued 'Grass'
```



## 2.1.4 Data Conversion

Conversion of data in Dataset is primarily between types for numerical features (int <-> float), and between categorical and numerical, and viceversa.

### Type conversion (int, float)

To convert between float and int:

```
pokemon.to_int('this_is_a_float_feature')
```

or

```
pokemon.to_float(['int_feature_1', 'int_feature_2'])
```

Again, you can pass a single name or a list of names between brackets.

### Categorical <-> Numerical

We can also convert numerical features to categorical and viceversa (when it makes sense), using:

```
pokemon.to_categorical('my_numerical_feature')
```

or

```
pokemon.to_numerical('my_categorical_feature')
```

In our case, it seems that the feature **Generation** could be considered as **categorical**. So, to convert it as a category, we better convert it first to int, to later convert it to a category. We can call both methods in the same line. We should also convert the target variable to the categorical type, since bool is often interpreted as a number (0/1).

```
[24]: pokemon.to_int('Generation').to_categorical(['Generation'])
      pokemon.summary()
```

```
Features Summary (all):
'Total'      : float64    Min.(180.) 1stQ(330.) Med.(450.) Mean(435.) 3rdQ(515.) Max.(780.)
↳
'HP'         : float64    Min.(1.0) 1stQ(50.0) Med.(65.0) Mean(69.2) 3rdQ(80.0) Max.(255.)
'Attack'     : float64    Min.(5.0) 1stQ(55.0) Med.(75.0) Mean(79.0) 3rdQ(100.) Max.(190.)
'Defense'    : float64    Min.(5.0) 1stQ(50.0) Med.(70.0) Mean(73.8) 3rdQ(90.0) Max.(230.)
'Sp. Atk'    : float64    Min.(10.0) 1stQ(49.7) Med.(65.0) Mean(72.8) 3rdQ(95.0) Max.(194.)
↳
'Sp. Def'    : float64    Min.(20.0) 1stQ(50.0) Med.(70.0) Mean(71.9) 3rdQ(90.0) Max.(230.)
↳
'Speed'      : float64    Min.(5.0) 1stQ(45.0) Med.(65.0) Mean(68.2) 3rdQ(90.0) Max.(180.)
'Generation': object      6 catego. '1'(166, 0.2075) '2'(165, 0.2062) '3'(160, 0.2000) '4
↳ '(121, 0.1512) ...
'Name'       : object      800 catego. 'Bulbasaur'(1, 0.0013) 'Ivysaur'(1, 0.0013)
↳ 'Venusaur'(1, 0.0013) 'VenusaurMega Venusaur'(1, 0.0013) ...
'Type 1'     : object      18 catego. 'Grass'(112, 0.1400) 'Fire'(98, 0.1225) 'Water'(70,
↳ 0.0875) 'Bug'(69, 0.0862) ...
'Type 2'     : object      18 catego. 'Poison'(97, 0.2343) 'nan'(35, 0.0845) 'Flying'(34,
↳ 0.0821) 'Dragon'(33, 0.0797) ...
'Legendary'  : bool        2 catego. 'False'(735, 0.9187) 'True'(65, 0.0813)
```

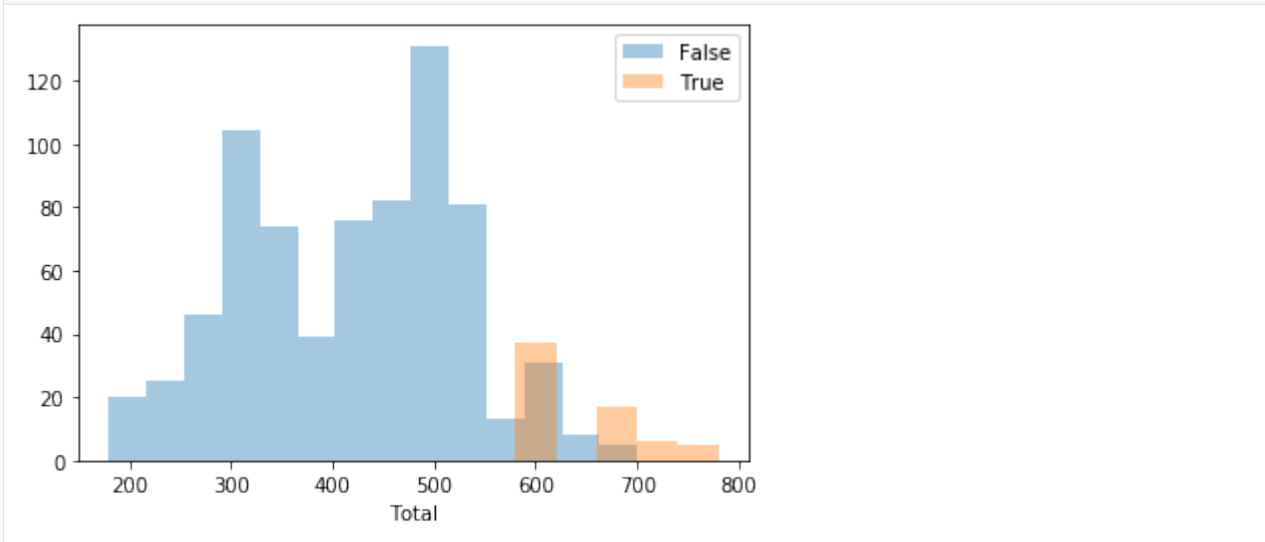
## 2.2 Histograms and density plots

### 2.2.1 Histograms

This function helps you to plot double histograms to compare feature-distributions for all possible target values. So, you **must set the target variable** before calling this method, or provide the name of the variable you want to compare your distribution against.

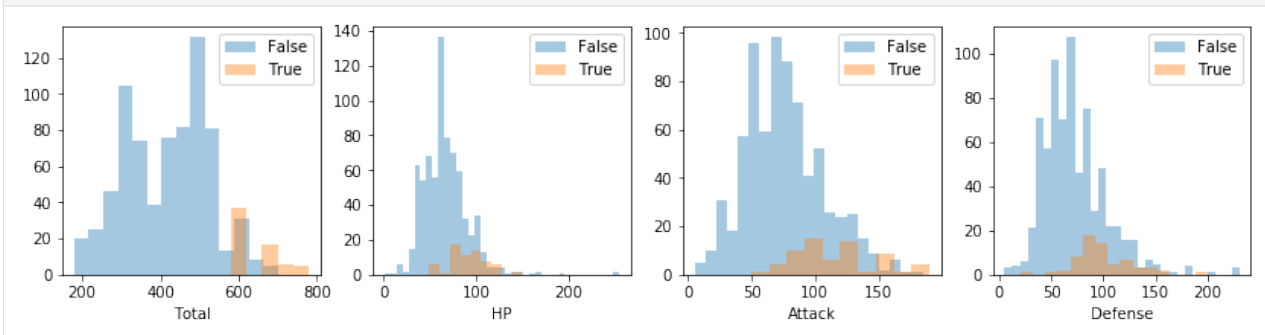
You can plot the histogram of any of the numerical variables with respect to the target variable to see what is its distribution, using (in this case we're plotting a feature called `Total` against the target variable –a binomial boolean feature previously set):

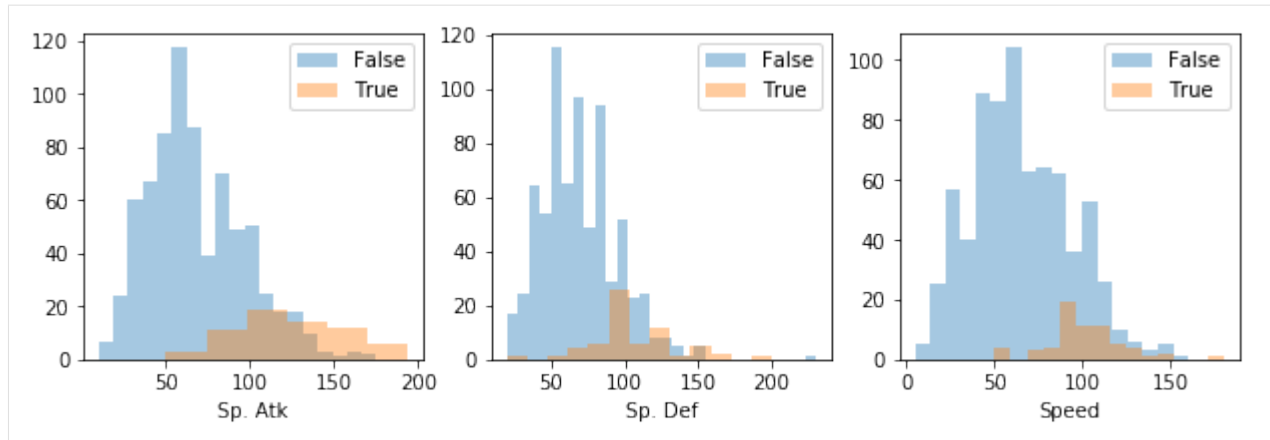
```
[25]: pokemon.plot_histogram('Total')
```



Or if you want, plot every numerical feature histogram):

```
[26]: pokemon.plot_histogram(pokemon.numerical_features)
```

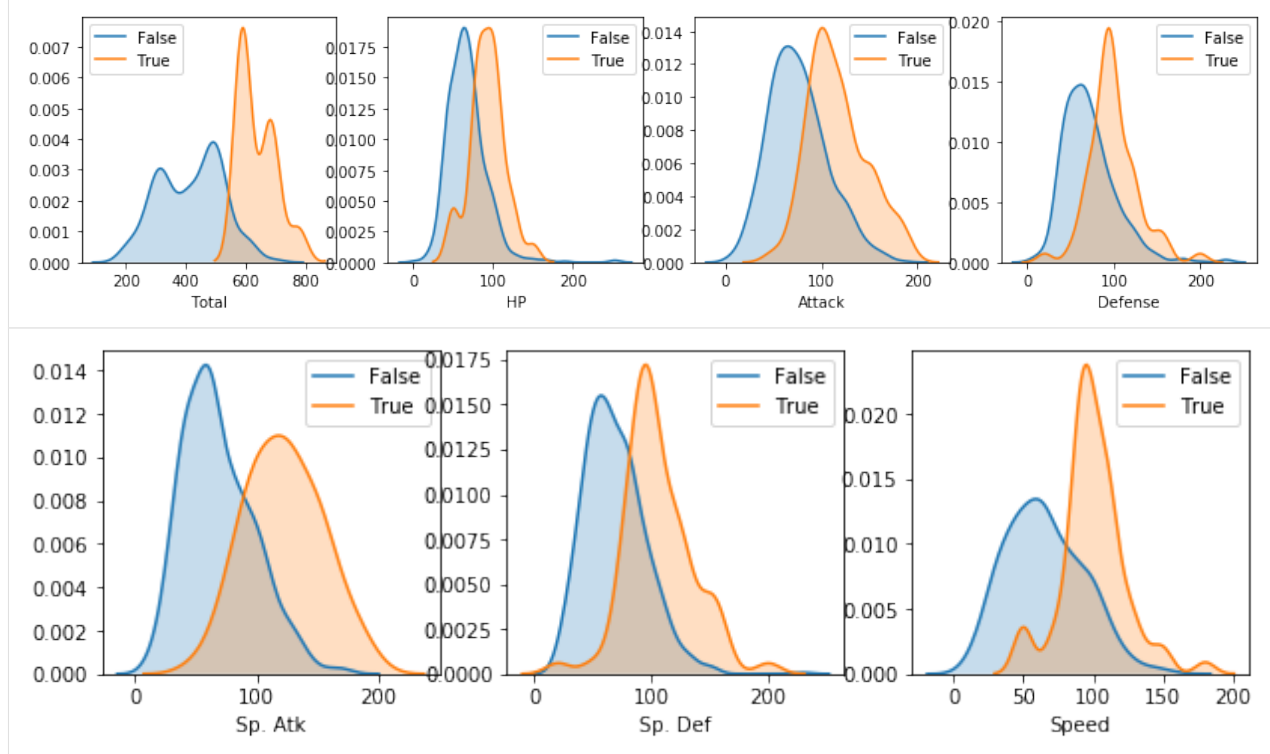




## 2.2.2 Density plots

Same applies to density plots (if no arguments are passed, all numerical features are considered)

[27]: `pokemon.plot_density()`



## 2.2.3 Features importance

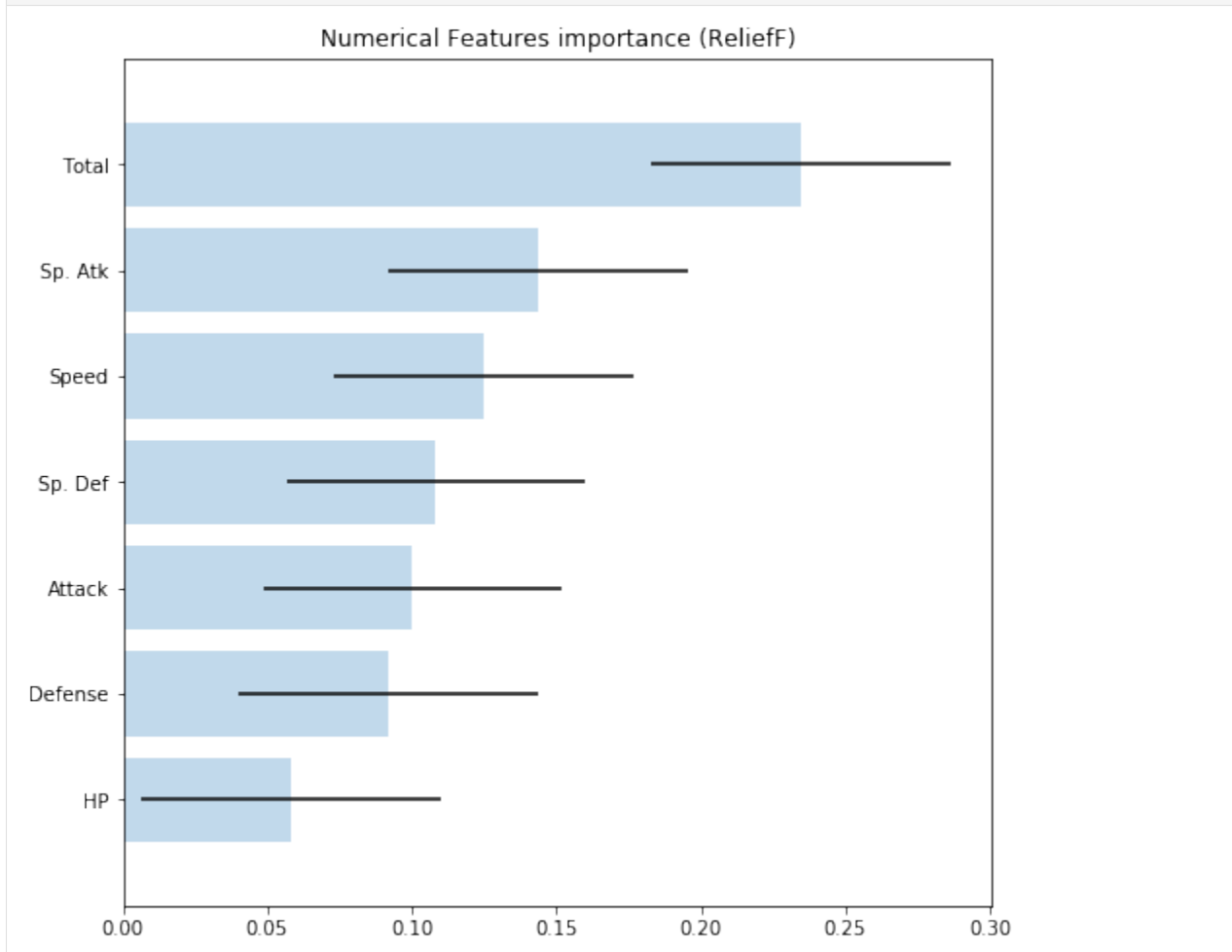
To extract features importance, Dataset uses the ReliefF algorithm. By calling the method `features_importance()` you obtain a Python dictionary with the name of every feature and its relative importance to predict the target variable.

```
[28]: pokemon.features_importance()
```

```
[28]: {'HP': 0.05809940944881894,  
      'Defense': 0.09187861111111111,  
      'Attack': 0.10025405405405405,  
      'Sp. Def': 0.10831636904761896,  
      'Speed': 0.12509607142857132,  
      'Sp. Atk': 0.14393648097826098,  
      'Total': 0.23443208333333318}
```

If you want to plot features importance, call `plot_importance()`:

```
[29]: pokemon.plot_importance()
```

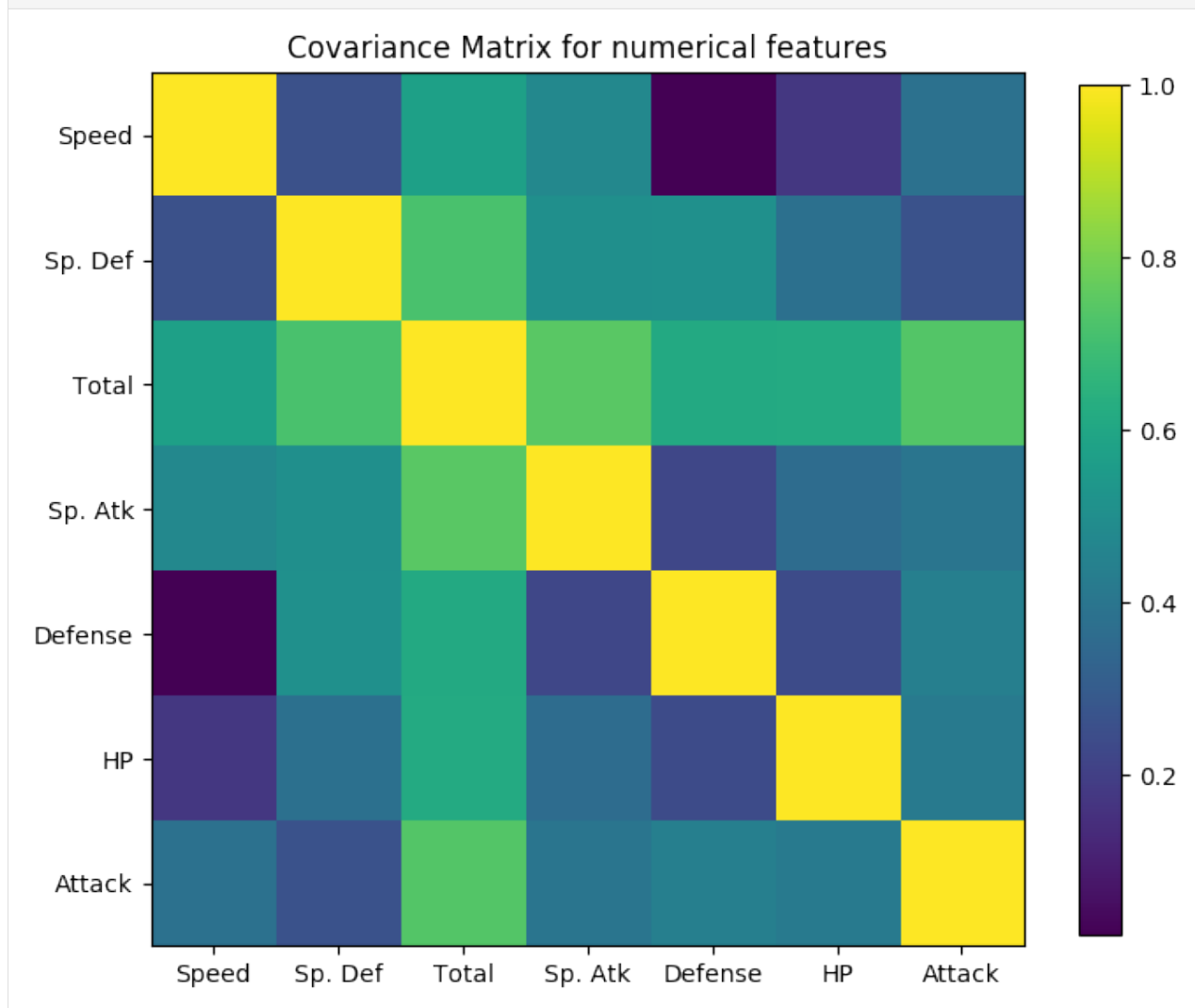


## 2.2.4 Covariance Matrix

Another useful plot is the covariance matrix. This time, Dataset library adds an interesting and convenient functionality, which is, grouping features with similar covariance together in the same plot. To do so, it uses a hierarchical dendrogram to determine what is the best possible order to reflect affinities between features.

To use it, simply type the following:

```
[30]: pokemon.plot_covariance()
```

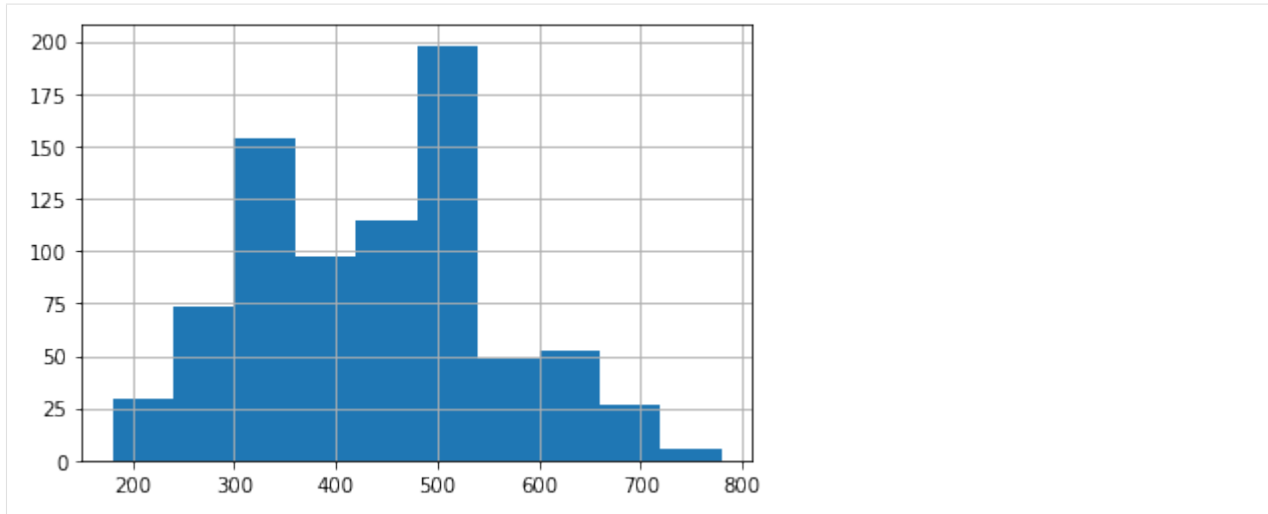


## 2.2.5 Native pandas plots

In case you want to access native pandas plotting functions, remember that you can access the entire dataframe by simply accessing the property `.features`, and from there, you can reference individual variables by its names. From that point, in order to access native pandas `hist()` for the feature `Total`, we use:

```
[28]: pokemon.features.Total.hist()
```

```
[28]: <matplotlib.axes._subplots.AxesSubplot at 0x129bcfe50>
```



## 2.3 Data cleaning

### 2.3.1 NA's

Identify and remove NA's is one of the basic capabilities in pandas. To ease that step, Dataset gives you info about features that contain NA's in the `describe()` method. If any, you can ask what features are presenting empty values by:

```
[29]: pokemon.nas()
```

```
[29]: ['Type 2']
```

and remove them, to check that everything worked fine (no need to say what features to fix).

```
[30]: pokemon.drop_na()  
pokemon.nas()      # <- this should return an empty array
```

```
[30]: []
```

### Replace NA

If you want to replace NA instead of removing, use `replace_na()`.

### 2.3.2 Outliers

To identify outliers, you can use the method `outliers()`. You can specify how many neighbours to consider when evaluating if a sample is an outlier (default value is 20). The method will tell you what indices in the dataset contain samples that might be outliers. From this point is your decision to remove them or not, and properly evaluate the effect of that eventual removal.

```
[31]: pokemon.outliers()
```

```
[31]: array([ 11,  20,  42,  49,  54, 105, 106, 110, 377])
```

```
[32]: pokemon.drop_samples(pokemon.outliers())
```

```
[32]: <dataset.dataset.Dataset at 0x10e75a220>
```

## 2.3.3 Correlation

Besides all the correlogram capabilities provided by scikit-learn and many other libraries, Dataset simply allows you to compute what is the correlation between features. No plot added, but all the relevant information is returned in a single call. The relevant parameter here is the threshold used to determine whether two features are correlated or not.

```
[33]: pokemon.correlated(threshold=0.7)
```

```
[33]: [('Total', 'Sp. Atk', 0.7448157606316953),
      ('Total', 'Sp. Def', 0.7409062430885279),
      ('Total', 'Attack', 0.7370599898479868),
      ('Total', 'HP', 0.7224040324193493)]
```

If you want to consider correlation with the target variable, you should unset the target variable (returning it back to the list of features) to compute its correlation with the remaining features (of the same type).

```
[34]: pokemon.unset_target().to_categorical('Legendary').categorical_correlated(threshold=0.1)
```

```
[34]: [('Generation', 'Type 2', 0.2806336628669287),
      ('Type 2', 'Legendary', 0.26714501722584993),
      ('Type 1', 'Type 2', 0.24543264607050674),
      ('Generation', 'Type 1', 0.23603346946402512),
      ('Type 1', 'Legendary', 0.19440503723011787),
      ('Generation', 'Legendary', 0.1889106739606925)]
```

```
[35]: pokemon.set_target('Legendary');
```

## 2.3.4 Under represented features

Dataset has a method to see if the values of some of the features are under represented. This situation occurs when one or several possible values from a feature are only present in a residual number of samples.

To discover if your dataset presents this anomaly, simply type:

```
[36]: pokemon.under_represented_features()
```

```
[36]: []
```

which in our case returns an empty array.

## Merge categories

In our case we don't have that situation. If that would be the case we can merge all under-represented categories together to have a balanced representation of all of them. To do so, we use

```
my_data.merge_categories(column='color', old_values=['grey', 'black'], new_value='dark')
```

to fusion values grey and black into a new category dark.

## Merge values

If we want to fusion values from numerical features, we should use:

```
my_data.merge_values(column='years', old_values=['2001', '2002'], new_value='2000')
```

## 2.4 Data transformation

### 2.4.1 One-hot encoding

Some categorical features are better transformed into numerical by performing a one-hot encoding. To do so, we call the method `onehot_encode()` specifying the list of features we want to convert. If no name is given, then all categorical variables are *onehot-encoded*.

In this case, the feature called `Generation` has been previously converted from numerical to categorical, but now we're transforming it into a dummified version, which will produce 6 new variables called `Generation_1` to `Generation_6`.

```
[37]: pokemon.onehot_encode('Generation').summary()
```

```
Features Summary (all):
'Attack'      : float64      Min.(20.0) 1stQ(60.0) Med.(80.0) Mean(83.0) 3rdQ(103.) Max.
↳ (190.)
'Defense'     : float64      Min.(15.0) 1stQ(55.0) Med.(75.0) Mean(78.2) 3rdQ(100.) Max.
↳ (180.)
'HP'          : float64      Min.(1.0) 1stQ(55.0) Med.(70.0) Mean(70.7) 3rdQ(85.0) Max.
↳ (150.)
'Name'        : object       405 categs. 'Bulbasaur'(1, 0.0025) 'Ivysaur'(1, 0.0025)
↳ 'Venusaur'(1, 0.0025) 'VenusaurMega Venusaur'(1, 0.0025) ...
'Sp. Atk'     : float64      Min.(20.0) 1stQ(50.0) Med.(70.0) Mean(77.3) 3rdQ(100.) Max.
↳ (180.)
'Sp. Def'     : float64      Min.(20.0) 1stQ(55.0) Med.(75.0) Mean(75.4) 3rdQ(95.0) Max.
↳ (154.)
'Speed'       : float64      Min.(10.0) 1stQ(50.0) Med.(70.0) Mean(70.9) 3rdQ(92.0) Max.
↳ (160.)
'Total'       : float64      Min.(190.) 1stQ(352.) Med.(474.) Mean(455.) 3rdQ(530.) Max.
↳ (780.)
'Type 1'      : object       18 categs. 'Grass'(51, 0.1259) 'Fire'(50, 0.1235) 'Bug'(37, 0.
↳ 0914) 'Normal'(36, 0.0889) ...
'Type 2'      : object       18 categs. 'Poison'(97, 0.2395) 'Flying'(33, 0.0815) 'Dragon
↳ '(32, 0.0790) 'Ground'(32, 0.0790) ...
'Generation_1': float64      Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.18) 3rdQ(0.0) Max.(1.0)
'Generation_2': float64      Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(1.0)
'Generation_3': float64      Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
```

(continues on next page)



(continued from previous page)

```
'Generation_4': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.16) 3rdQ(0.0) Max.(1.0)
'Generation_5': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
'Generation_6': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(1.0)
'Legendary'      : object    2 categs. 'False'(394, 0.9728) 'True'(11, 0.0272)
```

## 2.4.2 Discretize

In some other occasions what we need is to transform a numerical variable into a category by discretizing it, o binning. To illustrate this we will transform a numerical variable into a category by specifying ranges of values or bins to consider.

For example, the feature Speed (ranges between 10 and 160) could be discretized by considering only ranges, as follows:

```
[38]: pokemon.discretize('Speed', [(10, 60), (60, 110), (110, 150)],
                        category_names=['low', 'mid', 'high']).summary()
```

```
Features Summary (all):
'Attack'      : float64    Min.(20.0) 1stQ(60.0) Med.(80.0) Mean(83.0) 3rdQ(103.) Max.
↳ (190.)
'Defense'     : float64    Min.(15.0) 1stQ(55.0) Med.(75.0) Mean(78.2) 3rdQ(100.) Max.
↳ (180.)
'HP'          : float64    Min.(1.0) 1stQ(55.0) Med.(70.0) Mean(70.7) 3rdQ(85.0) Max.
↳ (150.)
'Name'        : object      405 categs. 'Bulbasaur'(1, 0.0025) 'Ivysaur'(1, 0.0025)
↳ 'Venusaur'(1, 0.0025) 'VenusaurMega Venusaur'(1, 0.0025) ...
'Sp. Atk'     : float64    Min.(20.0) 1stQ(50.0) Med.(70.0) Mean(77.3) 3rdQ(100.) Max.
↳ (180.)
'Sp. Def'     : float64    Min.(20.0) 1stQ(55.0) Med.(75.0) Mean(75.4) 3rdQ(95.0) Max.
↳ (154.)
'Speed'       : category    3 categs. 'low'(212, 0.5261) 'mid'(165, 0.4094) 'high'(26, 0.
↳ 0645)
'Total'       : float64    Min.(190.) 1stQ(352.) Med.(474.) Mean(455.) 3rdQ(530.) Max.
↳ (780.)
'Type 1'      : object      18 categs. 'Grass'(51, 0.1259) 'Fire'(50, 0.1235) 'Bug'(37, 0.
↳ 0914) 'Normal'(36, 0.0889) ...
'Type 2'      : object      18 categs. 'Poison'(97, 0.2395) 'Flying'(33, 0.0815) 'Dragon
↳ (32, 0.0790) 'Ground'(32, 0.0790) ...
'Generation_1': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.18) 3rdQ(0.0) Max.(1.0)
'Generation_2': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(1.0)
'Generation_3': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
'Generation_4': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.16) 3rdQ(0.0) Max.(1.0)
'Generation_5': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
'Generation_6': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(1.0)
'Legendary'   : object      2 categs. 'False'(394, 0.9728) 'True'(11, 0.0272)
```

### 2.4.3 Skewness

Another possibility is to fix the skewness of all those features who could present it. We can do it at once by simply calling `fix_skewness()`, or if we previously want to check what features present skewness, we can also call `skewed_features()`.

Let's apply it, first to know what features present skewness (if any):

```
[39]: pokemon.skewed_features()
```

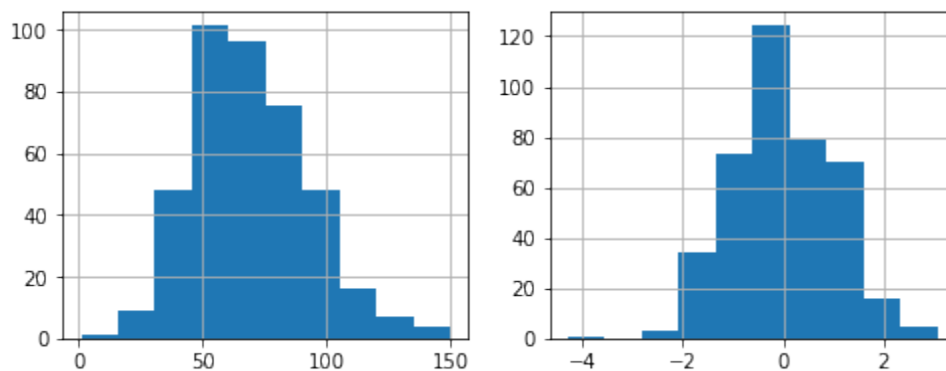
```
[39]: Generation_6    2.324424
      Generation_2    2.221659
      Generation_4    1.800833
      Generation_1    1.663678
      Generation_3    1.480842
      Generation_5    1.480842
      Sp. Atk         0.673112
      Attack          0.533859
      HP              0.498398
      Defense         0.471578
      Sp. Def         0.438338
      Total           0.070545
      dtype: float64
```

Let's fix skewness and plot the histogram before and after in the same plot.

```
[40]: plt.figure(figsize=(8,3))
      plt.subplot(121)
      pokemon.features['HP'].hist()

      # Fix skewness
      pokemon.fix_skewness()

      plt.subplot(122)
      pokemon.features['HP'].hist()
      plt.show()
```



## 2.4.4 Scale

It depends on the method that you use, but it is normally accepted that scaling your numeric features is a good practice. If you want to do so, you can easily do it by calling:

```
[41]: pokemon.scale(method='MinMaxScaler')
```

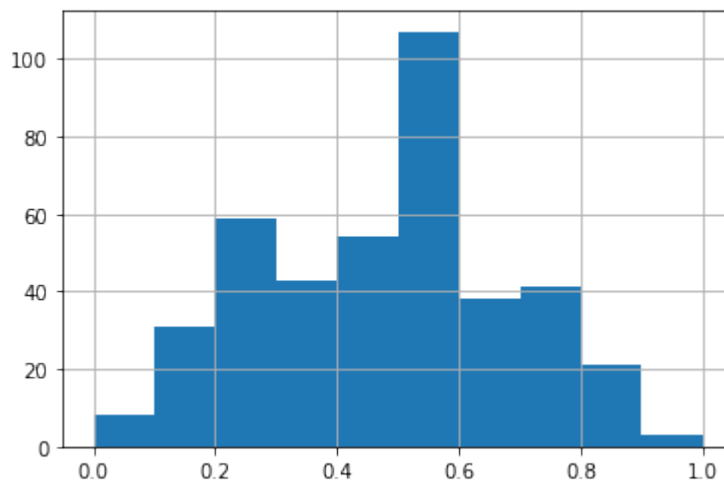
```
[41]: <dataset.dataset.Dataset at 0x10e75a220>
```

If you don't specify any parameters, scaling will be applied to numerical features, and the method will return the dataset with those features already scaled using `StandardScaler`. But you can also specify `MinMaxScaler` as method, if you want a different scaling method to be applied. The result is that numerical features range now between 0 and 1.

To confirm that scaling worked properly let's plot the histogram of the feature `Totals`, to see that the X-axis is now ranging between 0 and 1, instead of 0 and 800, as in the previous section plot.

```
[42]: pokemon.features.Total.hist()
```

```
[42]: <matplotlib.axes._subplots.AxesSubplot at 0x1297c5df0>
```



Speed is now a category that only presents three possible values (whose labels have been provided).

## 2.5 Feature selection

### 2.5.1 Information Gain

To compute the information gain provided by each categorical feature with respect to the target variable (must be set) then, you can use the method `information_gain()`. The result is a dictionary with key-value pairs, where each key corresponds to the categorical variables, and the value is the IG:

```
[43]: ig = pokemon.information_gain()
```

```
for k in ig:
    print('{:<7}: {:.2f}'.format(k, ig[k]))
```

```
Name    : 0.18
Speed   : 0.00
```

(continues on next page)

(continued from previous page)

```
Type 1 : 0.04
Type 2 : 0.03
```

From this info, it could be safe to remove the variable Speed, but must always test your hypothesis first.

## 2.5.2 Stepwise feature selection

We can force a feature selection process with our features by calling `stepwise_selection()`. We must be sure that the features used in the selection process are all numerical. If that is not the case, call `onehot_encode()` before using this method.

In our case, we must adapt a little bit our problem to suit the needs of `stepwise_selection()`. We need target variable to be numeric. To achieve that, we must unset the target variable `Legendary` in order to bring it back list of features. Then we call the method `onehot_encode()` (only works over the features, not the target variable), and then set the target again.

Remember that we can chain method calls:

```
[44]: pokemon.unset_target()
pokemon.onehot_encode('Legendary').drop_columns('Legendary_False')
pokemon.set_target('Legendary_True').summary()
```

```
Features Summary (all):
'Attack'      : float64    Min.(0.0) 1stQ(0.36) Med.(0.48) Mean(0.48) 3rdQ(0.61) Max.
↳ (1.0)
'Defense'     : float64    Min.(0.0) 1stQ(0.37) Med.(0.50) Mean(0.50) 3rdQ(0.64) Max.
↳ (1.0)
'Generation_1' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.18) 3rdQ(0.0) Max.(1.0)
'Generation_2' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(0.
↳ 99)
'Generation_3' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
'Generation_4' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.16) 3rdQ(0.0) Max.(1.0)
'Generation_5' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
'Generation_6' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(0.
↳ 99)
'HP'          : float64    Min.(0.0) 1stQ(0.49) Med.(0.58) Mean(0.58) 3rdQ(0.67) Max.
↳ (0.99)
'Name'        : object      403 categs. 'Bulbasaur'(1, 0.0025) 'Ivysaur'(1, 0.0025)
↳ 'Venusaur'(1, 0.0025) 'VenusaurMega Venusaur'(1, 0.0025) ...
'Sp. Atk'     : float64    Min.(0.0) 1stQ(0.37) Med.(0.51) Mean(0.52) 3rdQ(0.68) Max.
↳ (1.0)
'Sp. Def'     : float64    Min.(0.0) 1stQ(0.37) Med.(0.52) Mean(0.51) 3rdQ(0.66) Max.
↳ (1.0)
'Speed'       : category    3 categs. 'low'(212, 0.5261) 'mid'(165, 0.4094) 'high'(26,
↳ 0.0645)
'Total'       : float64    Min.(0.0) 1stQ(0.30) Med.(0.51) Mean(0.47) 3rdQ(0.60) Max.
↳ (1.0)
'Type 1'      : object      18 categs. 'Grass'(51, 0.1266) 'Fire'(49, 0.1216) 'Bug'(36,
↳ 0.0893) 'Normal'(36, 0.0893) ...
'Type 2'      : object      18 categs. 'Poison'(96, 0.2382) 'Flying'(33, 0.0819) 'Dragon
↳ (32, 0.0794) 'Ground'(32, 0.0794) ...
'Legendary_True': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.02) 3rdQ(0.0) Max.(1.0)
```

We remove then the column called `Legendary_False` as the column `Legendary_True` already behaves numerically as we want (0.0 means False, and 1.0 means True).

Now, we let the stepwise algorithm to decide what features could be safe to remove:

```
[45]: pokemon.stepwise_selection()
```

```
Considering only numerical features
```

```
[45]: ['Generation_3', 'Generation_4']
```

In a one-liner, we drop the columns/features selected by the stepwise algorithm and then printout the summary.

```
[46]: pokemon.drop_columns(pokemon.stepwise_selection()).summary()
```

```
Considering only numerical features
```

```
Features Summary (all):
```

```
'Attack'      : float64    Min.(0.0) 1stQ(0.36) Med.(0.48) Mean(0.48) 3rdQ(0.61) Max.
↳(1.0)
'Defense'     : float64    Min.(0.0) 1stQ(0.37) Med.(0.50) Mean(0.50) 3rdQ(0.64) Max.
↳(1.0)
'Generation_1' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.18) 3rdQ(0.0) Max.(1.0)
'Generation_2' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(0.
↳99)
'Generation_5' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.20) 3rdQ(0.0) Max.(1.0)
'Generation_6' : float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.12) 3rdQ(0.0) Max.(0.
↳99)
'HP'          : float64    Min.(0.0) 1stQ(0.49) Med.(0.58) Mean(0.58) 3rdQ(0.67) Max.
↳(0.99)
'Name'        : object      403 categs. 'Bulbasaur'(1, 0.0025) 'Ivysaur'(1, 0.0025)
↳'Venusaur'(1, 0.0025) 'VenusaurMega Venusaur'(1, 0.0025) ...
'Sp. Atk'     : float64    Min.(0.0) 1stQ(0.37) Med.(0.51) Mean(0.52) 3rdQ(0.68) Max.
↳(1.0)
'Sp. Def'     : float64    Min.(0.0) 1stQ(0.37) Med.(0.52) Mean(0.51) 3rdQ(0.66) Max.
↳(1.0)
'Speed'       : category    3 categs. 'low'(212, 0.5261) 'mid'(165, 0.4094) 'high'(26,
↳0.0645)
'Total'       : float64    Min.(0.0) 1stQ(0.30) Med.(0.51) Mean(0.47) 3rdQ(0.60) Max.
↳(1.0)
'Type 1'      : object      18 categs. 'Grass'(51, 0.1266) 'Fire'(49, 0.1216) 'Bug'(36,
↳0.0893) 'Normal'(36, 0.0893) ...
'Type 2'      : object      18 categs. 'Poison'(96, 0.2382) 'Flying'(33, 0.0819) 'Dragon
↳'(32, 0.0794) 'Ground'(32, 0.0794) ...
'Legendary_True': float64    Min.(0.0) 1stQ(0.0) Med.(0.0) Mean(0.02) 3rdQ(0.0) Max.(1.0)
```

## 2.6 Dataset Split

Dataset adds a method to split your dataset according to the specified proportions between training and test. The method is called `split()`, and accepts as optional parameter the percentage to be assigned to the test set.

We normally split specifying the seed used by the random number generator. If you plan to repeat the split process a number of times within a CV process, you need to change the seed accordingly.

```
[47]: X, y = pokemon.split(seed=1)
```

What we obtain in `X` and `y`, are two objects that inside contain the training and test splits, as pandas DataFrames. To access them, we simply write:

```
X.train
X.test
y.train
y.test
```

Let's check it out:

```
[48]: X.train.head(3)
```

```
[48]:      Attack  Defense  Generation_1  Generation_2  Generation_5  \
185  0.575444  0.335548             0.0             0.0             0.0
227  0.778264  0.611023             0.0             0.0             0.0
245  0.633856  0.439686             0.0             0.0             0.0

      Generation_6      HP      Name  Sp. Atk  Sp. Def  Speed  \
185             0.0  0.426154      Anorith  0.265005  0.328587  mid
227             0.0  0.556204  LopunnyMega  0.393294  0.672391  high
245             0.0  0.661813      Toxicroak  0.609585  0.453749  mid

      Total  Type 1  Type 2
185  0.309899   Rock    Bug
227  0.688447  Normal  Fighting
245  0.541394  Poison  Fighting
```

```
[49]: y.test.head(3)
```

```
[49]:      Legendary_True
360             0.0
62              0.0
374             0.0
```

Depending on the ML method used, you can directly pass `X.train` and `y.train` as DataFrames, or in some other cases to you need to pass the numpy array of values. In that case you simple use:

```
X.train.values, y.train.values
```

Same applies to test subsets.

## THE API DOCUMENTATION

If you are looking for information on a specific function, or method, this part of the documentation is for you.

### 3.1 dataset package

#### 3.1.1 Submodules

#### 3.1.2 dataset.correlations module

`dataset.correlations.conditional_entropy(x, y)`

Calculates the conditional entropy of  $x$  given  $y$ :  $S(x|y)$

Wikipedia: <[https://en.wikipedia.org/wiki/Conditional\\_entropy](https://en.wikipedia.org/wiki/Conditional_entropy)>

**Parameters**

- **x** – list / NumPy ndarray / Pandas Series A sequence of measurements
- **y** – list / NumPy ndarray / Pandas Series A sequence of measurements

**Returns** float

`dataset.correlations.convert(data, to)`

`dataset.correlations.correlation_ratio(categories, measurements)`

Calculates the Correlation Ratio (sometimes marked by the greek letter Eta) for categorical-continuous association. Answers the question - given a continuous value of a measurement, is it possible to know which category is it associated with? Value is in the range  $[0,1]$ , where 0 means a category cannot be determined by a continuous measurement, and 1 means a category can be determined with absolute certainty.

Wikipedia: [https://en.wikipedia.org/wiki/Correlation\\_ratio](https://en.wikipedia.org/wiki/Correlation_ratio)

**Parameters**

- **categories** – list / NumPy ndarray / Pandas Series A sequence of categorical measurements
- **measurements** – list / NumPy ndarray / Pandas Series A sequence of continuous measurements

**Returns** float in the range of  $[0,1]$

`dataset.correlations.cramers_v(x, y)`

Calculates Cramer's V statistic for categorical-categorical association. Uses correction from Bergsma and Wicher, Journal of the Korean Statistical Society 42 (2013): 323-328. This is a symmetric coefficient:  $V(x,y) = V(y,x)$

Original function taken from: <https://stackoverflow.com/a/46498792/5863503> Wikipedia: [https://en.wikipedia.org/wiki/Cram%C3%A9r%27s\\_V](https://en.wikipedia.org/wiki/Cram%C3%A9r%27s_V)

**Parameters**

- **x** – list / NumPy ndarray / Pandas Series A sequence of categorical measurements
- **y** – list / NumPy ndarray / Pandas Series A sequence of categorical measurements

**Returns** float in the range of [0,1]

`dataset.correlations.theils_u(x, y)`

Calculates Theil's U statistic (Uncertainty coefficient) for categorical-categorical association. This is the uncertainty of x given y: value is on the range of [0,1] - where 0 means y provides no information about x, and 1 means y provides full information about x. This is an asymmetric coefficient:  $U(x,y) \neq U(y,x)$

Wikipedia: [https://en.wikipedia.org/wiki/Uncertainty\\_coefficient](https://en.wikipedia.org/wiki/Uncertainty_coefficient)

**Parameters**

- **x** – list / NumPy ndarray / Pandas Series A sequence of categorical measurements
- **y** – list / NumPy ndarray / Pandas Series A sequence of categorical measurements

**Returns** float in the range of [0,1]

### 3.1.3 dataset.dataset module

This is the package dataset.

**class** `dataset.dataset.Dataset(data_location=None, data_frame=None, *args, **kwargs)`

Bases: object

This class allows a simpler representation of the dataset used to build a model in class. It allows to load a remote CSV by providing an URL to the initialization method of the object, and work on the most common tasks related to data preparation and feature engineering.:

```
my_data = Dataset(URL)

my_data = Dataset.from_dataframe(my_dataframe)
```

**add\_columns(new\_features)**

Add a Series as a new column to the dataset.

**Parameters new\_features** – A pandas Series object or a DataFrame with the data to be added to the Dataset. It must contain a valid name not present in the Dataset already.

Examples:

```
my_data.add_column(my_series)
my_data.add_column(pandas.Series().values)
my_data.add_column(my_dataframe)
```

**aggregate(col\_list, new\_column, operation='sum', drop\_columns=True)**

Perform an arithmetic operation on the given columns, and places the result on a new column, removing the original ones.

**Parameters**

- **col\_list** – the list of columns over which the operation is done
- **new\_column** – the name of the new column to be generated from the operation



- **drop\_columns** – whether remove the columns used to perform the aggregation
- **operation** – the operation to be done over the column values for each row. Examples: 'sum', 'diff', 'max', etc. By default, the operation is the sum of the values.

**Returns** The Dataset object

Example:

If we want to sum the values of column1 and column2 into a new column called 'column3', we use:

```
my_data.aggregate(['column1', 'column2'], 'column3', 'sum')
```

As a result, my\_data will remove column1 and column2, and the operation will be the sum of the values, as it is the default operation.

**all** = None

**categorical** = None

**categorical\_correlated**(threshold=0.9)

Generates a correlation matrix for the categorical variables in dataset Calculates Cramer's V statistic for categorical-categorical association. Uses correction from Bergsma and Wicher, Journal of the Korean Statistical Society 42 (2013): 323-328. This is a symmetric coefficient:  $V(x,y) = V(y,x)$  Original function taken from:

<https://stackoverflow.com/a/46498792/5863503>

**Wikipedia:** [http://en.wikipedia.org/wiki/Cram%C3%A9r%27s\\_V](http://en.wikipedia.org/wiki/Cram%C3%A9r%27s_V)

**Parameters threshold** – Limit from which correlations is considered high.

**Returns** The list of categorical variables with HIGH correlation and the correlation matrix

**categorical\_dtypes** = ['bool', 'object', 'string', 'category']

**property categorical\_features**

**property categorical\_features\_na**

**correlated**(threshold=0.9)

Return the features that are highly correlated to with other variables, either numerical or categorical, based on the threshold. For numerical variables Spearman correlation is used, for categorical crammers\_v.

**Parameters threshold** – correlation limit above which features are considered highly correlated.

**Returns** the list of features that are highly correlated, and should be safe to remove.

**describe**(feature\_name=None, inline=False)

Wrapper. Calls the proper feature description method, depending on whether the feature is numerical or categorical. If no arguments are passed, the description of the entire dataset is provided.

**Parameters**

- **feature\_name** – The feature to be described. Default value is None, which implies that all features are described.
- **inline** – whether the output is multiple lines or inline. This is used when describing from `summary()` function or from a console or cell.

**Returns** The string, only when inline=True, that contains the description.

**TODO: Implement a limit of characters for each line that is printed** out in the screen, so that when reaching that limit '...' is printed.

**describe\_dataset()**

Printout the metadata information collected when calling the metainfo() method.

**Returns** nothing

**discretize(column, bins, category\_names=None)**

Makes a feature, which is normally numerical, categorical by binning its contents into the specified buckets.

**Args:** column: The name of the feature to be binned bins: the list of bins as an array of values of the form [(15, 20), (20, 25), (25, 30), (30, 35), (35, 40)]

**category\_names:** An array with names or values we want for our new categories. If None a simple array with ordinal number of the category is used. In the example above, it should be an array from 1 .. 5.

Returns: The dataset modified

Example:

```
# Variable "x3" contains the number of sons of a person as an
# integer ranging between values 0 and 10. We want to convert
# that numerical value into a categorical one with a list
# of (say) 4 possible values, for the number of sons within
# given ranges:

my_data.discretize('x3',
                   [(0, 2), (2, 4), (4, 6), (6, 8)], [1, 2, 3, 4])
```

**drop\_columns(columns\_list)**

Drop one or a list of columns from the dataset.

**Parameters columns\_list** – An array-type expression with the names of the columns to be removed from the Dataset. In case a single string is passed, it will be considered the name of a single columns to be dropped.

Examples:

```
my_data.drop_columns('column_name')
my_data.drop_columns(['column1', 'column2', 'column3'])
```

**drop\_na()**

Drop samples with NAs from the features. If any value is infinite or -infinite, it is converted to NA, and removed also.

Examples:

```
my_data.drop_na()
```

**Returns** object

**drop\_samples(index\_list)**

Remove the list of samples from the dataset.

**Parameters index\_list** – The list of indices in the DataFrame to be removed from the features and the target DataFrames.

**Returns** self

**property** feature\_names

**features** = None

**features\_importance**(num\_features=None, num\_neighbors=None, abs\_imp=False)

Computes NUMERICAL features importance, using the ReliefF algorithm as implemented in the *rebate* library.

**Args:** num\_features: The nr of features we want to display num\_neighbors: The nr of neighbors to consider when computing the

features importance

abs\_imp: if True, importance is displayed taking the ABS()

**Returns:** A sorted dictionary with the feature names and their importance.

**fix\_skewness**(feature\_names=None, return\_series=False)

Ensures that the numerical features in the dataset, fit into a normal distribution by applying the Yeo-Johnson transform. If not already scaled, they're scaled as part of the process.

**Parameters**

- **feature\_names** – Features to be fixed. If not specified, all numerical features are examined.
- **return\_series** – Return the normalized series

**Returns** The subset fitted to normal distribution, or None

**classmethod** from\_dataframe(df)

**property** incomplete\_features

**information\_gain**()

Computes the information gain between each categorical and target variable.

Examples:

```
my_data.information_gain()
Name      : 0.18
Speed     : 0.00
Type 1    : 0.04
Type 2    : 0.03
```

**Returns:** A dictionary with the IG value for each categorical feature name

**keep\_columns**(to\_keep)

Keep only one or a list of columns from the dataset.

**Parameters** to\_keep – A string or array-like expression indicating the columns to be kept in the Dataset. The columns not in the list of names passed are dropped.

Example:

```
my_data.keep_columns('column_name')
my_data.keep_columns(['column1', 'column2', 'column3'])
```

**merge\_categories**(column, old\_values, new\_value)

Merge a subset of categories present in one of the columns into a new single category. This is normally done when this list of cats is not enough representative.

**Parameters**

- **column** – The column with the categories to be merged
- **old\_values** – The list of categories to be merged
- **new\_value** – The resulting new category after the merge.

**Returns** self.

Example:

```
my_data.merge_categories(column='color',
                        old_values=['grey', 'black'],
                        new_value='dark')
```

**merge\_values**(*column, old\_values, new\_value*)

Same method as ‘merge\_categories’ but for numerical values. Merge a subset of values present in one of the columns into a new single category. This is normally done when this list of values is not enough representative.

**Parameters**

- **column** – The column with the values to be merged
- **old\_values** – The list of values to be merged
- **new\_value** – The resulting new value after the merge.

**Returns** self.

Example:

```
my_data.merge_values(column='years',
                    old_values=['2001', '2002'],
                    new_value='2000')
```

**meta = None**

**meta\_tags** = ['all', 'numerical', 'categorical', 'complete', 'numerical\_na', 'categorical\_na', 'features', 'target']

**names**(*what='all'*)

Returns a the names of the columns of the dataset for which the arg *what* is specified. If it is a list, it returns those feature names in the list, And if it is a keyword from: ‘all’, ‘categorical’, ‘categorical\_na’, ‘numerical’, ‘numerical\_na’, ‘complete’, then the list of features is extracted from the metainformation of the dataset.

**Parameters** **what** – Possible values are

- all: (Default) Include very feature, including the target
- numerical: Only numerical features
- categorical: Only categorical features
- complete: Only features without NA
- numerical\_na: Numerical features with NA
- categorical\_na: Categorical features with NA
- features: Only features, NOT the target variable.
- target: Only the target variable.

**nas()**

Returns the list of features that present NA entries

**Returns** the list of feature names presenting NA

**property num\_features**

**property num\_samples**

**numerical = None**

**numerical\_correlated(threshold=0.9)**

Build a correlation matrix between all the features in data set

**Parameters threshold** – Threshold beyond which considering high correlation. Default is 0.9

**Returns** The list of columns that are highly correlated and could be drop out from dataset.

**property numerical\_features**

**property numerical\_features\_na**

**onehot\_encode(feature\_names=None)**

Encodes the categorical features in the dataset, with OneHotEncode

**Parameters feature\_names** – column or list of columns to be one-hot encoded. The only restriction is that the target variable cannot be specified in the list of columns and therefore, cannot be onehot encoded. Default = all categorical features in dataset.

**Returns** self

Example:

```
# Encodes a single column named 'my_column_name'
my_data.onehot_encode('my_column_name')

# Encodes 'col1' and 'col2'
my_data.onehot_encode(['col1', 'col2'])

# Encodes all categorical features in the dataset
my_data.onehot_encode(my_data.names('categorical'))
```

or:

```
my_data.onehot_encode()
```

**outliers(n\_neighbors=20)**

Find outliers, using LOF criteria, from the numerical features. Returns a list of indices where outliers are present

**Parameters n\_neighbors** – Number of neighbors to use by default for kneighbors queries. If n\_neighbors is larger than the number of samples provided, all samples will be used.

**# TODO Implement a simple set of methods to select from in order to** detect outliers.

**static plot\_correlation\_matrix(corr\_matrix)**

**plot\_covariance()**

Plots the covariance matrix as explained by scikit contributor Andreas Mueller in Columbia lectures, ordering and grouping (numerical) features with higher correlation.

**Returns:** None

**plot\_density**(*feature\_names=None, category=None*)

Double density plot(s) between feature(s) and a reference category.

**Parameters**

- **feature\_names** – The name of a feature(s) in the dataset.
- **category** – The name of the reference category we want to represent the double density plot against. If None, then the target variable is used.

**Returns** None

Example:

```
# represent multiple density plots, one per unique value of the
# target
my_data.plot_density(my_feature)

# represent double density plots, one per unique value of the
# categorical feature 'my_feature2'
my_data.plot_density(my_feature1, my_feature2)

# Plot double density plots for all numerical features.
my_data.plot_density(my_data.numerical_features)

# or
my_data.plot_density()
```

**plot\_histogram**(*feature\_names=None, category=None*)

Double histogram plot between a feature and a reference category.

**Parameters**

- **feature\_names** – The name(s) of the feature(s) in the dataset.
- **category** – The name of the reference category we want to represent the double density plot against. If None, then the target variable is used.

**Returns** None

Example:

```
# represent multiple density plots, one per unique value of the
# target
my_data.plot_double_hist(my_feature)

# represent double density plots, one per unique value of the
# categorical feature 'my_feature2'
my_data.double_hist(my_feature1, my_feature2)

# or
my_data.plot_density()
```

**plot\_importance**(*num\_features=None, num\_neighbors=None, abs\_imp=False*)

Plots the NUMERICAL features importance, using the ReliefF algorithm as implemented in the *rebate* library.

**Args:**

**num\_features:** The nr of features we want to display. Default is all features.

**num\_neighbors:** The nr of neighbors to consider when computing the features importance. Default is 20.

**abs\_imp:** if True, importance is displayed taking the ABS() Default value is False.

**Returns:** None

**replace\_na**(*column, value*)

Replace any NA occurrence from the column or list of columns passed by the value passed as second argument.

**Parameters**

- **column** – Column name or list of column names from which to replace NAs with the value passes in the second argument
- **value** – value to be used as replacement

**Returns** the object.

**samples\_matching**(*value=None, feature=None*)

Return the a list with the indexes of those samples matching a given criteria. The match can be set on target variable, or any other column name.

**Args:** value: feature:

**Returns:** A list with the index values of those samples matching.

Examples:

```
my_data.samples_matching('red')
```

returns the indices of those samples whose *target* matches the value *red*.

```
my_data.samples_matching(75, 'column_3')
```

returns the indices of those samples whose feature *column\_3* values 75.

**scale**(*features\_of\_type='numerical', method='StandardScaler', return\_series=False*)

Scales numerical features in the dataset, unless the parameter 'what' specifies any other subset selection primitive. The method to be used is the scikit learn StandardScaler.

Examples:

```
# scale all my numerical features
my_data.scale()
```

**Parameters**

- **features\_of\_type** – Subset selection primitive
- **method** – 'StandardScaler', 'MinMaxScaler'

**Returns** the subset scaled.

**select**(*what*)

Returns a subset of the columns of the dataset. *what* specifies what subset of features to return If it is a list, it returns those feature names in the list, And if it is a keyword from: 'all', 'categorical', 'categorical\_na', 'numerical', 'numerical\_na', 'complete', 'features', 'target', then the list of features is extracted from the metainformation of the dataset.

**Parameters** **what** – Possible values are

- all: (Default) Include very feature, including the target

- **numerical**: Only numerical features
- **categorical**: Only categorical features
- **complete**: Only features without NA
- **numerical\_na**: Numerical features with NA
- **categorical\_na**: Categorical features with NA
- **features**: Only features, NOT the target variable.
- **target**: Only the target variable.

**Returns** Reference to the columns specified.

**set\_target**(*target\_name*)

Set the target variable for this dataset. This will create a new property of the object called 'target' that will contain the target column of the dataset, and that column will be removed from the list of features.

**Parameters** **target\_name** – The name of the column we want to be set as the target variable for this dataset.

Example:

```
my_data.set_target('SalePrice')
```

**skewed\_features**(*threshold=0.75, fix=False, return\_series=True*)

Returns the list of numerical features that present skewness. This method optionally can fix detected skewness whose ABS is greater than the threshold passed, using BoxCox method.

**Parameters**

- **threshold** – The limit over which considering that the `skew()` return value is considered a skewed feature.
- **fix** – (Default: False) Boolean indicating whether or not fixing the skewed features. If True, those with values above the threshold will be fixed using BoxCox.
- **return\_series** – (Default: True) Boolean indicating whether returning the features (pandas DataFrame) that present skewness.

**Returns** A pandas Series with the features and their skewness

**split**(*seed=1024, test\_size=0.2, validation\_split=False*)

From an Dataset, produce splits (with or without validation) for training and test. The objects of type `Split` will only contain properties with the names `train` or `test` to reference the different splits.

**Parameters**

- **seed** – The seed to be used to generate the random split.
- **test\_size** – The test size as a percentage of the base dataset.
- **validation\_split** – Boolean indicating whether it is also needed to generate a third split for validation purposes, same size as the `test_size`.

**Returns** The X and y objects that contain the splits.

Example:

```
# Generate the splits (80-20)
X, y = my_data.split()
```

(continues on next page)



(continued from previous page)

```
# Create an instance of the model, and use the training set to
# fit it, and the test set to score it.
model = LinearRegression()
model.fit(X.train, y.train)
model.score(X.test, y.test)
```

**stepwise\_selection**(*initial\_list=None, threshold\_in=0.01, threshold\_out=0.05, verbose=False*)

Perform a forward/backward feature selection based on p-value from statsmodels.api.OLS Your features must be all numerical, so be sure to onehot\_encode them before calling this method. Always set threshold\_in < threshold\_out to avoid infinite looping. All features involved must be numerical and types must be float. Target variable must also be float. You can convert it back to a categorical type after calling this method.

#### Parameters

- **initial\_list** – list of features to start with (column names of X)
- **threshold\_in** – include a feature if its p-value < threshold\_in
- **threshold\_out** – exclude a feature if its p-value > threshold\_out
- **verbose** – whether to print the sequence of inclusions and exclusions

**Returns** List of selected features

Example:

```
my_data.stepwise_selection()
```

See <[https://en.wikipedia.org/wiki/Stepwise\\_regression](https://en.wikipedia.org/wiki/Stepwise_regression)> for the details

Taken from: <<https://datascience.stackexchange.com/a/24823>>

**summary**(*what='all'*)

Printout a summary of each feature.

**Parameters** **what** – Possible values are

- all: (Default) Include very feature, including the target
- numerical: Only numerical features
- categorical: Only categorical features
- complete: Only features without NA
- numerical\_na: Numerical features with NA
- categorical\_na: Categorical features with NA
- features: Only features, NOT the target variable.
- target: Only the target variable.

**Returns** N/A

**table**(*what='all', max\_width=80*)

Print a tabulated version of the list of elements in a list, using a max\_width display (default 80).

#### Parameters

- **what** – Possible values are
  - all: (Default) Include very feature, including the target

- numerical: Only numerical features
  - categorical: Only categorical features
  - complete: Only features without NA
  - numerical\_na: Numerical features with NA
  - categorical\_na: Categorical features with NA
  - features: Only features, NOT the target variable.
  - target: Only the target variable.
- **max\_width** – The max\_width used in the display.

**Returns** None

**target** = None

**to\_categorical**(*to\_convert*)

Convert the specified column or columns to categories

**Parameters** **to\_convert** – column or column list to be converted

**Returns** object

**to\_float**(*to\_convert=None*)

Convert a column or list of columns to float values. The columns must be numerical.

**Args:**

**to\_convert: the column name or list of column names that we want** to convert. If this argument is empty, then every numerical feature in the dataset is converted.

Returns: The dataset

Example:

```
my_data.to_float(my_data.numerical_features)

# which is equivalent to::
my_data.to_float()

# We can also specify a single or multiple features::
my_data.to_float('feature_15')
my_data.to_float(['feature_15', 'feature_21'])
```

**to\_int**(*to\_convert=None*)

Convert a column or list of columns to integer values. The columns must be numerical

**Args:**

**to\_convert: the column name or list of column names that we want** to convert. If none specified, all numerical columns are converted to int type.

Returns: The dataset

Example:

```
my_data.to_int(my_data.numerical_features)

# which is equivalent to::
my_data.to_int()
```

(continues on next page)

(continued from previous page)

```
# We can also specify a single or multiple features::
my_data.to_int('feature_15')
my_data.to_int(['feature_15', 'feature_21'])
```

**to\_numerical**(*to\_convert*)

Convert the specified column or columns to numbers

**Parameters** *to\_convert* – column name or list of column names to be converted

**Returns** object

**TODO: It must be possible to perform label encoding if specified.** For example, I might want to convert a target variable with strings valued “Yes” and “No” to type “category” or to type “int” with values 1 and 0.

**under\_represented\_features**(*threshold=0.98*)

Returns the list of categorical features with unrepresented categories or a clear unbalance between the values that can take.

**Parameters** *threshold* – The upper limit of the most represented category of the feature.

**Returns** the list of features that with unrepresented categories.

**unset\_target**()

Undo the *set\_target()* operation. The feature *target\_name* returns to the DataFrame with the rest of the features.

Example:

```
my_data.unset_target()
```

### 3.1.4 dataset.split module

**class** dataset.split.Split(*splits*)

Bases: object

This class represents a split from a dataset, it will assign each dataframe partition passed as argument to a different attribute of the class: ‘train’, ‘test’ (and ‘validation’). The class method ‘split’ performs the splitting of the dataframe passed, according to the parameters passed.

Example:

```
from src import split
X, Y = split.Split(my_dataframe, my_target_column)
split_name = ['train', 'test', 'validation']
```

### 3.1.5 Module contents

Dataset module for machine learning basic dataframe manipulation and feature engineering tasks

## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### d

- `dataset`, [40](#)
- `dataset.correlations`, [27](#)
- `dataset.dataset`, [28](#)
- `dataset.split`, [39](#)





## A

`add_columns()` (*dataset.dataset.Dataset* method), 28  
`aggregate()` (*dataset.dataset.Dataset* method), 28  
`all` (*dataset.dataset.Dataset* attribute), 29

## C

`categorical` (*dataset.dataset.Dataset* attribute), 29  
`categorical_correlated()` (*dataset.dataset.Dataset* method), 29  
`categorical_dtypes` (*dataset.dataset.Dataset* attribute), 29  
`categorical_features` (*dataset.dataset.Dataset* property), 29  
`categorical_features_na` (*dataset.dataset.Dataset* property), 29  
`conditional_entropy()` (in module *dataset.correlations*), 27  
`convert()` (in module *dataset.correlations*), 27  
`correlated()` (*dataset.dataset.Dataset* method), 29  
`correlation_ratio()` (in module *dataset.correlations*), 27  
`cramers_v()` (in module *dataset.correlations*), 27

## D

`dataset`  
    module, 40  
`Dataset` (class in *dataset.dataset*), 28  
`dataset.correlations`  
    module, 27  
`dataset.dataset`  
    module, 28  
`dataset.split`  
    module, 39  
`describe()` (*dataset.dataset.Dataset* method), 29  
`describe_dataset()` (*dataset.dataset.Dataset* method), 30  
`discretize()` (*dataset.dataset.Dataset* method), 30  
`drop_columns()` (*dataset.dataset.Dataset* method), 30  
`drop_na()` (*dataset.dataset.Dataset* method), 30  
`drop_samples()` (*dataset.dataset.Dataset* method), 30

## F

`feature_names` (*dataset.dataset.Dataset* property), 31  
`features` (*dataset.dataset.Dataset* attribute), 31  
`features_importance()` (*dataset.dataset.Dataset* method), 31  
`fix_skewness()` (*dataset.dataset.Dataset* method), 31  
`from_dataframe()` (*dataset.dataset.Dataset* class method), 31

## I

`incomplete_features` (*dataset.dataset.Dataset* property), 31  
`information_gain()` (*dataset.dataset.Dataset* method), 31

## K

`keep_columns()` (*dataset.dataset.Dataset* method), 31

## M

`merge_categories()` (*dataset.dataset.Dataset* method), 31  
`merge_values()` (*dataset.dataset.Dataset* method), 32  
`meta` (*dataset.dataset.Dataset* attribute), 32  
`meta_tags` (*dataset.dataset.Dataset* attribute), 32  
module  
    *dataset*, 40  
    *dataset.correlations*, 27  
    *dataset.dataset*, 28  
    *dataset.split*, 39

## N

`names()` (*dataset.dataset.Dataset* method), 32  
`nas()` (*dataset.dataset.Dataset* method), 32  
`num_features` (*dataset.dataset.Dataset* property), 33  
`num_samples` (*dataset.dataset.Dataset* property), 33  
`numerical` (*dataset.dataset.Dataset* attribute), 33  
`numerical_correlated()` (*dataset.dataset.Dataset* method), 33  
`numerical_features` (*dataset.dataset.Dataset* property), 33  
`numerical_features_na` (*dataset.dataset.Dataset* property), 33

## O

`onehot_encode()` (*dataset.dataset.Dataset method*), 33  
`outliers()` (*dataset.dataset.Dataset method*), 33

## P

`plot_correlation_matrix()` (*dataset.dataset.Dataset static method*), 33  
`plot_covariance()` (*dataset.dataset.Dataset method*), 33  
`plot_density()` (*dataset.dataset.Dataset method*), 33  
`plot_histogram()` (*dataset.dataset.Dataset method*), 34  
`plot_importance()` (*dataset.dataset.Dataset method*), 34

## R

`replace_na()` (*dataset.dataset.Dataset method*), 35

## S

`samples_matching()` (*dataset.dataset.Dataset method*), 35  
`scale()` (*dataset.dataset.Dataset method*), 35  
`select()` (*dataset.dataset.Dataset method*), 35  
`set_target()` (*dataset.dataset.Dataset method*), 36  
`skewed_features()` (*dataset.dataset.Dataset method*), 36  
`Split` (*class in dataset.split*), 39  
`split()` (*dataset.dataset.Dataset method*), 36  
`split_name` (*dataset.split.Split attribute*), 39  
`stepwise_selection()` (*dataset.dataset.Dataset method*), 37  
`summary()` (*dataset.dataset.Dataset method*), 37

## T

`table()` (*dataset.dataset.Dataset method*), 37  
`target` (*dataset.dataset.Dataset attribute*), 38  
`theils_u()` (*in module dataset.correlations*), 28  
`to_categorical()` (*dataset.dataset.Dataset method*), 38  
`to_float()` (*dataset.dataset.Dataset method*), 38  
`to_int()` (*dataset.dataset.Dataset method*), 38  
`to_numerical()` (*dataset.dataset.Dataset method*), 39

## U

`under_represented_features()` (*dataset.dataset.Dataset method*), 39  
`unset_target()` (*dataset.dataset.Dataset method*), 39