**Data pre-processing with Python/Pandas.**

Preprocessing involves the following aspects:

* Missing values
* Data standardization
* Outliers’ detection

**Installing Anaconda (For your laptop)**

In this class, we will be using the google colaboratory online environment to perform all our tutorial tasks.

However, you can install other tools on your own machines, such as anaconda and other notebooks.

It can be difficult to install a Python machine-learning environment on some platforms. Python itself must be installed first, and then there are many packages to install, and it can be confusing for beginners.

You can easily set up a Python machine-learning development environment using Anaconda.

You will have a working Python environment to begin learning, practicing, and developing machine learning.

This task has been already covered in the Data Mining and Machine Learning Software Requirements Folder on the blackboard. For a recap, please visit the following link [Machine Learning with Anaconda](https://machinelearningmastery.com/setup-python-environment-machine-learning-deep-learning-anaconda/)

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn.

# scipy

import scipy

print('scipy: %s' % scipy.\_\_version\_\_)

# numpy

import numpy

print('numpy: %s' % numpy.\_\_version\_\_)

# matplotlib

import matplotlib

print('matplotlib: %s' % matplotlib.\_\_version\_\_)

# pandas

import pandas

print('pandas: %s' % pandas.\_\_version\_\_)

# statsmodels

import statsmodels

print('statsmodels: %s' % statsmodels.\_\_version\_\_)

# scikit-learn

import sklearn

print('sklearn: %s' % sklearn.\_\_version\_\_)

**Task 1: Loading your dataset in google colaboratory**

To launch google colaboratory go to <https://colab.research.google.com/> you will need to login with your university user ID. Then open a **new notebook**.

To import your dataset into the google colab environment expand **table of content,** then select **files.** Click **upload to this session storage,** then selectyour dataset from your machine directory system and click **open.** Click on the **“kebab” menu** next to the data set and **copy path;** this is the new path for the dataset in the google colab platform.From here, we no longer use the path on your machine’s directory system.

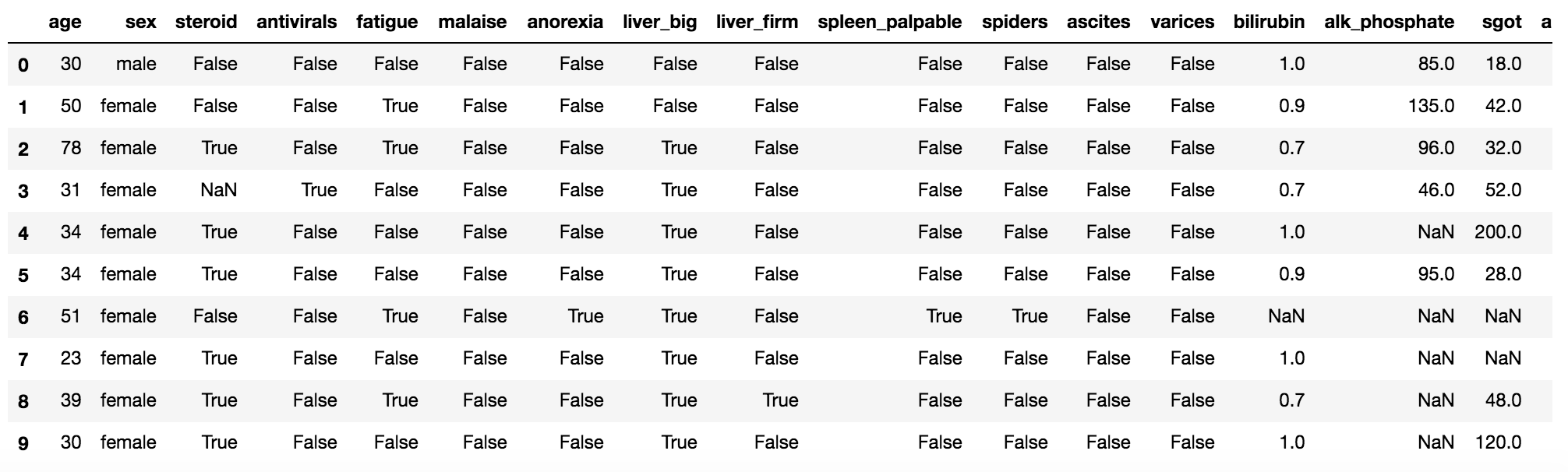
**Task 2: Dealing with Missing Values**

**Import data**

In this tutorial, we will use the dataset related to Hepatitis, which can be downloaded from [this link](https://datahub.io/machine-learning/hepatitis#pandas).

Firstly, import data using the pandas library and convert them into a dataframe. Through the head(10) method we print only the first 10 rows of the dataset.

**import** pandas **as** pd  
df **=** pd.read\_csv('/content/hepatitis.csv')  
df.head(10)



**Identify missing values**

We note that the dataset presents some problems. For example, the column email is not available for all the rows. In some cases it presents the NaN value, which means that the value is missing.

In order to check whether our dataset contains missing values, we can use the function isna(), which returns if an cell of the dataset if NaN or not. Then we can count how many missing values there are for each column.

data.isna().sum()

which gives the following output:

age 0  
sex 0  
steroid 1  
antivirals 0  
fatigue 1  
malaise 1  
anorexia 1  
liver\_big 10  
liver\_firm 11  
spleen\_palpable 5  
spiders 5  
ascites 5  
varices 5  
bilirubin 6  
alk\_phosphate 29  
sgot 4  
albumin 16  
protime 67  
histology 0  
class 0  
dtype: int64

Now we can count the percentage of missing values for each column, simply by dividing the previous result by the length of the dataset (len(df)) and multiplying per 100.

data.isna().sum()**/**len(data)**\***100

which gives the following output:

age 0.000000  
sex 0.000000  
steroid 0.645161  
antivirals 0.000000  
fatigue 0.645161  
malaise 0.645161  
anorexia 0.645161  
liver\_big 6.451613  
liver\_firm 7.096774  
spleen\_palpable 3.225806  
spiders 3.225806  
ascites 3.225806  
varices 3.225806  
bilirubin 3.870968  
alk\_phosphate 18.709677  
sgot 2.580645  
albumin 10.322581  
protime 43.225806  
histology 0.000000  
class 0.000000  
dtype: float64

When dealing with missing values, different alternatives can be applied:

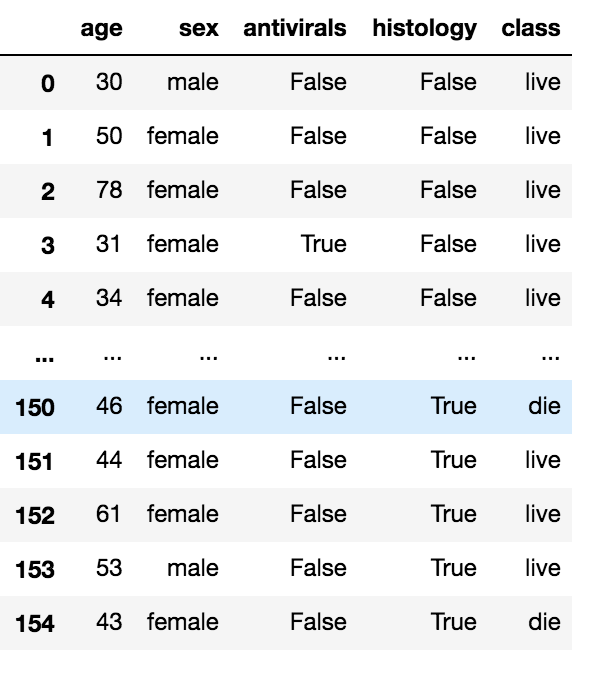
* check the source, for example, by contacting the data source to correct the missing values
* drop missing values
* replace the missing value with a value
* leave the missing value as it is.

Drop missing values

Dropping missing values can be one of the following alternatives:

* remove rows having missing values
* remove the whole column containing missing values We can use the dropna() by specifying the axis to be considered. If we set axis = 0 we drop the entire row, if we set axis = 1 we drop the whole column. If we apply the function df.dropna(axis=0) 80 rows of the dataset remain. If we apply the function df.dropna(axis=1), only the columns age, sex, antivirals, histology and class remain. However, removed values are not applied to the original dataframe, but only to the result. We can use the argument inplace=True in order to store changes in the original dataframe df (df.dropna(axis=1,inplace=True)).

data.dropna(axis**=**1)



As an alternative, we can specify only the column on which the dropping operation must be applied. In the following example, only missing rows related to the column liver\_big are considered. This can be achieved through the subset parameter, which permits to specify the subset of columns where to apply the dropping operation.

data.dropna(subset**=**['liver\_big'],axis**=**0,inplace**=True**)

Now we can check whether there are still missing values for the column

data.isna().sum()/len(data)\*100

Another alternative involves the dropping of columns where a certain percentage of not-null values is available. This can be achieved through the thresh parameter. In the following example we keep only columns where there are at least the 80% of not null values.

data.dropna(thresh**=**0.8**\***len(data),axis**=**1,inplace**=True**)

**Replace missing values**

A good strategy when dealing with missing values involves their replacement with another value. Usually, the following strategies are adopted:

* for numerical values replace the missing value with the average value of the column
* for categorial values replace the missing value with the most frequent value of the column
* use other functions

In order to replace missing values, three functions can be used: fillna(), replace() and interpolate(). The fillna() function replaces all the NaN values with the value passed as argument. For example, for numerical values, all the NaN values in the numeric columns could be replaced with the average value. In order to list the type of a column, we can use the attribute dtypes as follows:

data.dtypes

which gives the following output:

age int64  
sex object  
steroid object  
antivirals bool  
fatigue object  
malaise object  
anorexia object  
liver\_big object  
liver\_firm object  
spleen\_palpable object  
spiders object  
ascites object  
varices object  
bilirubin float64  
alk\_phosphate float64  
sgot float64  
albumin float64  
histology bool  
class object  
dtype: object

**Numeric columns**

Firstly, we select numeric columns.

**import** numpy **as** np  
numeric **=** data.select\_dtypes(include**=**np.number)  
numeric\_columns **=** numeric.columns

Then, we fill the NaN values of numeric columns with the average value, given by the data.mean() function.

data[numeric\_columns] **=** data[numeric\_columns].fillna(data.mean())

Now, we can check whether the NaN values in numeric columns have been removed.

data.isna().sum()**/**len(data)**\***100

which gives the following output:

age 0.000000  
sex 0.000000  
steroid 0.689655  
antivirals 0.000000  
fatigue 0.000000  
malaise 0.000000  
anorexia 0.000000  
liver\_big 0.000000  
liver\_firm 0.689655  
spleen\_palpable 0.689655  
spiders 0.689655  
ascites 0.689655  
varices 0.689655  
bilirubin 0.000000  
alk\_phosphate 0.000000  
sgot 0.000000  
albumin 0.000000  
histology 0.000000  
class 0.000000  
dtype: float64

**Categorial columns**

We note that in dtypes the categorial columns are described as objects. Thus we can select the object columns. We would like to consider only boolean columns. However the object type includes also the column class, which is a string. We select all the object columns, and then we remove from them the column class. Then we can convert the type of the result to bool.

boolean\_columns **=** data.select\_dtypes(include**=**np.object).columns.tolist()  
boolean\_columns.remove('class')  
data[boolean\_columns] **=** data[boolean\_columns].astype('bool')

Now we can replace all the missing values for booleans with the most frequent value. We can use the mode() function to calculate the most frequent value. We use the fillna() function to replace missing values, but we could use also the replace(old\_value,new\_value) function.

data[boolean\_columns].fillna(df.mode())

Now our dataset does not contain any missing value.

data.isna().sum()**/**len(data)**\***100

which gives the following output:

age 0.0  
sex 0.0  
steroid 0.0  
antivirals 0.0  
fatigue 0.0  
malaise 0.0  
anorexia 0.0  
liver\_big 0.0  
liver\_firm 0.0  
spleen\_palpable 0.0  
spiders 0.0  
ascites 0.0  
varices 0.0  
bilirubin 0.0  
alk\_phosphate 0.0  
sgot 0.0  
albumin 0.0  
histology 0.0  
class 0.0  
dtype: float64

**Interpolation**

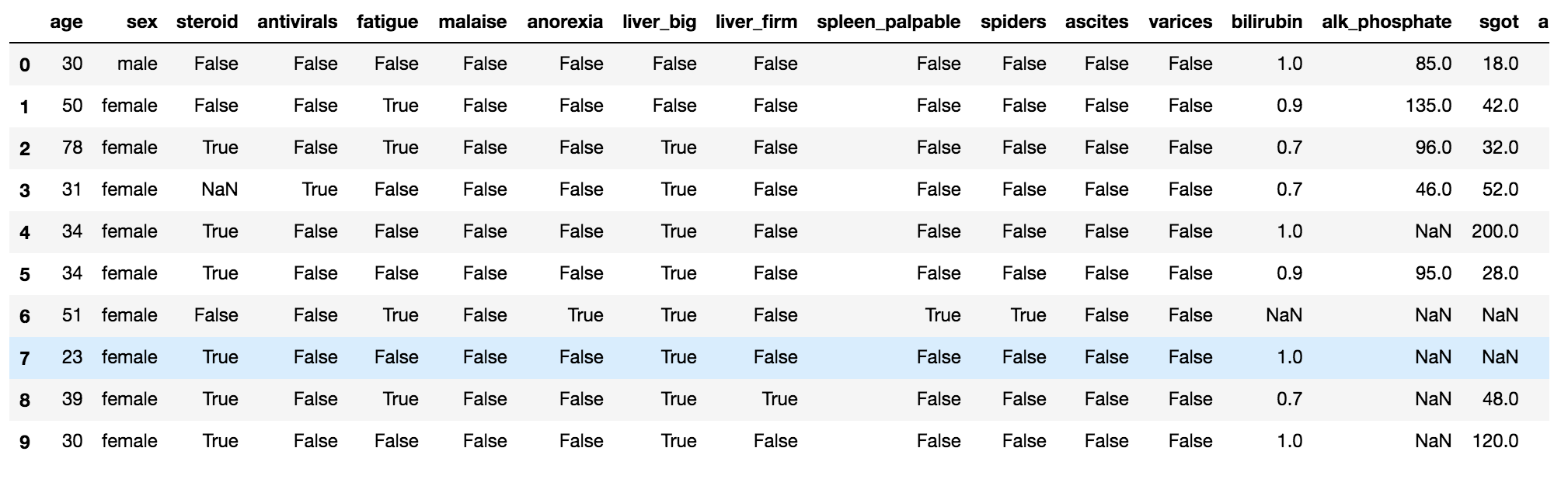
Another solution to replace missing values involves the usage of other functions, such as linear interpolation. In this case, for example, we could replace a missing value over a column, with the interpolation between the previous and the next ones. This can be achieved through the use of the interpolate() function.

Since we have already managed all the missing values, we reload the dataset.

data **=** pd.read\_csv('hepatitis.csv')  
data.isna().sum()**/**len(data)**\***100

We select only numeric columns.

numeric **=** data.select\_dtypes(include**=**np.number)  
numeric\_columns **=** numeric.columns  
data.head(10)



Now we can apply the interpolate() function to numeric columns, by setting also the limit direction to forward. This means that the linear interpolation is applied starting from the first row until the last one.

data[numeric\_columns] **=** data[numeric\_columns].interpolate(method **=**'linear', limit\_direction **=**'forward')

For example, in line 6 the column bilirubin, which was NaN before the interpolation, now assumes the value 0.95, which is the interpolation between 0.90 (line 4) and 1.00 (line 6).

data.head(10)



**Task 3: Dealing with Outliers**

Detecting the outliers

Outliers can be detected using visualization, implementing mathematical formulas on the dataset, or using the statistical approach. All of these are discussed below.

### 1. **Visualization**

**Example 1: Using Box Plot**

It captures the summary of the data effectively and efficiently with only a simple box and whiskers. Boxplot summarizes a sample data using 25th, 50th, and 75th percentiles. One can get insights(quartiles, median, and outliers) into the dataset by just looking at its boxplot.

import sklearn

from sklearn.datasets import load\_boston

import pandas as pd

import matplotlib.pyplot as plt

# Load the dataset

bos\_hou1 = load\_boston()

# Create the dataframe

column\_name1 = bos\_hou1.feature\_names

df\_boston1 = pd.DataFrame(bos\_hou1.data)

df\_boston1.columns = column\_name1

df\_boston1.head()

# Box Plot

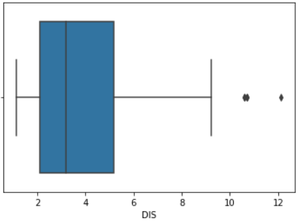
import seaborn as sns

sns.boxplot(df\_boston1['DIS'])

# Position of the Outlier

print(np.where(df\_boston1['DIS']>10))

In the above graph, we can clearly see that values above 10 are acting as outliers.



**Using [ScatterPlot](https://www.geeksforgeeks.org/matplotlib-pyplot-scatter-in-python/).**

It is used when you have paired numerical data, or when your dependent variable has multiple values for each reading independent variable, or when trying to determine the relationship between the two variables. In the process of utilizing the scatter plot, one can also use it for outlier detection.

To plot the scatter plot one requires two variables that are somehow related to each other. So here, ‘Proportion of non-retail business acres per town’ and ‘Full-value property-tax rate per $10,000’ are used whose column names are “INDUS” and “TAX” respectively.

**# Scatter plot**

fig, ax = plt.subplots(figsize = (18,10))

ax.scatter(df\_boston1['INDUS'], df\_boston1['TAX'])

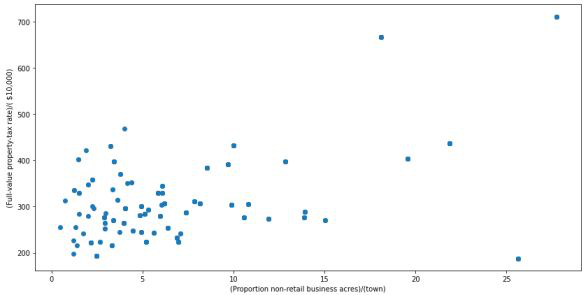
**# x-axis label**

ax.set\_xlabel('(Proportion non-retail business acres)/(town)')

**# y-axis label**

ax.set\_ylabel('(Full-value property-tax rate)/( $10,000)')

plt.show()



Looking at the graph can summarize that most of the data points are in the bottom left corner of the graph but there are few points that are exactly opposite that is the top right corner of the graph. Those points in the top right corner can be regarded as Outliers.

Using approximation can say all those data points that are x>20 and y>600 are outliers. The following code can fetch the exact position of all those points that satisfy these conditions.

print(np.where((df\_boston1['INDUS']>20) & (df\_boston1['TAX']>600)))

**3. IQR (Inter Quartile Range)**

IQR (Inter Quartile Range) Inter Quartile Range approach to finding the outliers is the most trusted approach used in the research field.

*IQR = Quartile3 – Quartile1*

To define the outlier base value is defined above and below datasets normal range namely Upper and Lower bounds, define the upper and the lower bound (1.5\*IQR value is considered):

*upper = Q3 -1.5\*IQR*

*lower = Q1 – 1.5\*IQR*

## **Removing the outliers**

For removing the outlier, one must follow the same process of removing an entry from the dataset using its exact position in the dataset because in all the above methods of detecting the outliers end result is the list of all those data items that satisfy the outlier definition according to the method used.

*dataframe.drop( row\_index, inplace = True)*

The above code can be used to drop a row from the dataset given the row\_indexes to be dropped. Inplace =True is used to tell python to make the required change in the original dataset. row\_index can be only one value or list of values or NumPy array but it must be one dimensional.

**Example:**

*df\_boston.drop(lists[0],inplace = True)*

**Full Code**: Detecting the outliers using IQR and removing them

# Importing

import sklearn

from sklearn.datasets import load\_boston

import pandas as pd

import numpy as np

# Load the dataset

bos\_hou = load\_boston()

# Create the dataframe

column\_name = bos\_hou.feature\_names

df\_boston = pd.DataFrame(bos\_hou.data)

df\_boston.columns = column\_name

df\_boston.head()

''' Detection '''

# IQR

Q1 = np.percentile(df\_boston['DIS'], 25,

interpolation = 'midpoint')

Q3 = np.percentile(df\_boston['DIS'], 75,

interpolation = 'midpoint')

IQR = Q3 - Q1

print("Old Shape: ", df\_boston.shape)

# Upper bound

upper = np.where(df\_boston['DIS'] >= (Q3+1.5\*IQR))

# Lower bound

lower = np.where(df\_boston['DIS'] <= (Q1-1.5\*IQR))

''' Removing the Outliers '''

df\_boston.drop(upper[0], inplace = True)

df\_boston.drop(lower[0], inplace = True)

print("New Shape: ", df\_boston.shape)

**Task 3: Normalisation**

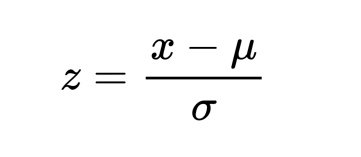
The scaling of Features is an essential step in modelling the algorithms with the datasets. The data that is usually used for the purpose of modelling is derived through various means, such as:

* Questionnaire
* Surveys
* Research, etc.

So, the data obtained contains features of various dimensions and scales altogether. Different scales of the data features affect the modelling of a dataset adversely. It leads to a biased outcome of predictions in terms of misclassification error and accuracy rates. Thus, it is necessary to Scale the data prior to modelling.

**This is when standardization comes into the picture.**

*Standardization is a scaling technique wherein it makes the data scale-free by converting the statistical distribution of the data into the below format:*



By this, the entire data set scales with a zero mean and unit variance, altogether. Let us now try to implement the concept of Standardization in the upcoming sections.

**Steps:**

1. Import the necessary libraries required. We have imported the sklearn library to use the StandardScaler function.
2. Load the dataset. Here we have used the **Boston housing dataset**
3. Set an object to the StandardScaler() function.
4. Apply the function onto the dataset using the fit\_transform() function.

from sklearn.preprocessing import StandardScaler

dataset = df\_boston1

scaler= StandardScaler()

# standardization

scaled\_dataset = scaler.fit\_transform(dataset)

print(scaled\_dataset)