Data Representation in Scikit-Learn
Machine learning is about creating models from data. It is important then to understand how data can be represented in order to be understood by the computer. The best way to think about data within Scikit-Learn is in terms of tables of data.

Data as table
A basic table is a two-dimensional grid of data, in which the rows represent individual elements of the dataset, and the columns represent quantities related to each of these elements.

Features matrix
The table layout makes the information to be represented as a two dimensional numerical array or matrix, which is called the *features matrix*. By convention, this features matrix is often stored in a variable named X. Generally, features matrix is assumed to be two-dimensional, with shape [n_samples, n_features], and is most often contained in a NumPy array or a Pandas DataFrame, though some Scikit-Learn models also accept SciPy sparse matrices.
The samples (rows) always refer to the individual objects described by the dataset. The sample might be, a person, a document, an image, a sound file, a video, or anything else that can be described with a set of quantitative measurements.
The features (columns) refer to the distinct observations that describe each sample in a quantitative manner. Features are generally real-valued, but may be Boolean or discrete-valued in some cases.

Target array
In addition to the feature matrix X, you also need to work with a *label* or *target* array that by convention is called *y*. This target array is usually one dimensional, with length n_samples, and can be contained in a NumPy array or Pandas Series. It may have continuous numerical values, or discrete classes/labels. While some Scikit-Learn estimators do handle multiple target values in the form of a two-dimensional [n_samples, n_targets] target array, you will be working with the a one-dimensional target array.
Notice that the distinguishing feature of the target array is the quantity that needs to be *predicted from the data* (i.e dependent variable).

Scikit-Learn's Estimator API
Every machine learning algorithm in Scikit-Learn is implemented via the Estimator API, which provides a consistent interface for a wide range of machine learning applications.

Basics of the API
The following are the steps commonly used in the Scikit-Learn estimator API:

i). Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
ii). Choose model hyperparameters by instantiating the class with desired values.
iii). Arrange data into a features matrix and target vector
iv). Fit the model to your data by calling the *fit()* method of the model instance.
v). Apply the model to new data:
- For supervised learning, you predict labels for unknown data using the predict() method.
- For unsupervised learning, you transform or infer properties of the data using the transform() or predict() method.

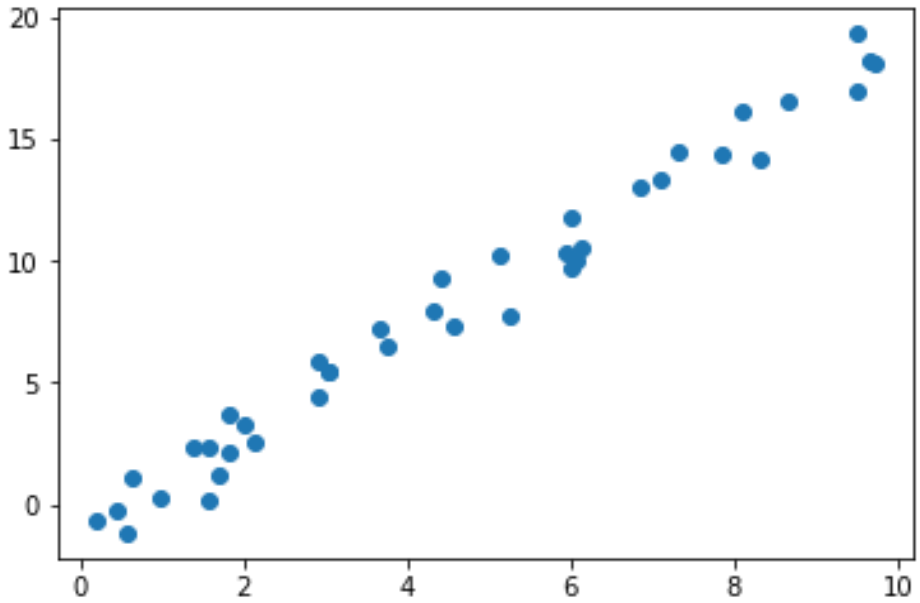
vi). Check the results of model fitting to know whether the model is satisfactory

Simple linear regression
Linear regression involves coming up with a straight-line fit to data. A straight-line fit is a model of the form $y = ax + b$ where a is commonly known as the *slope*, and b is commonly known as the *intercept*.

Example
Consider a simple linear regression that involves fitting a line to x, y data. Use the following data sample:

```
In[1]: import matplotlib.pyplot as plt
import numpy as np
rng = np.random.RandomState(42)
x = 10 * rng.rand(40)
y = 2 * x - 1 + rng.randn(40)
print (x)
print (y)
plt.scatter(x, y);
```

```
Out[1]:

[3.74540119 9.50714306 7.31993942 5.98658484 1.5601864  1.5599452
 0.58083612 8.66176146 6.01115012 7.08072578 0.20584494 9.69909852
 8.32442641 2.12339111 1.81824967 1.8340451  3.04242243 5.24756432
 4.31945019 2.9122914  6.11852895 1.39493861 2.92144649 3.66361843
 4.56069984 7.85175961 1.99673782 5.14234438 5.92414569 0.46450413
 6.07544852 1.70524124 0.65051593 9.48885537 9.65632033 8.08397348
 3.04613769 0.97672114 6.84233027 4.40152494]
[ 6.47730515 16.9565752  14.46242375  9.75232603  2.3292364   0.16022028
 -1.16651381 16.52038415 11.76076681 13.33281984 -0.7039584  18.09709335
 14.17033083  2.52693801  2.17586057  3.72521242  5.42846315  7.73208848
  7.96298434  4.43950052 10.56013589  2.4015535   5.87389249  7.25851698
  7.28218216 14.39430685  3.32473907 10.2602339  10.36911714 -0.25665072
 10.04456206  1.21427585  1.11355768 19.33395077 18.24063054 16.17147986
  5.45391141  0.30832253 13.04605614  9.34108644]
```



With this data, you can use the steps outlined for the Scikit-Learn estimator API above:

i). Choose a class of model.

In Scikit-Learn, every class of model is represented by a Python class. E.g., for a simple linear regression model, you import the linear regression class as shown below:

```
In[2]: from sklearn.linear_model import LinearRegression
```

ii). Choose model hyper parameters.

Depending on the model class you are working with, you might need to consider one or more of the following options:

- Would you like to fit for the offset (i.e., intercept)?
  Use **`fit_intercept`** (True by default) that decides whether to calculate the intercept b_0 (True) or consider it equal to zero (False).
- Would you like the model to be normalized?
  Use **`normalize`** (False by default) that decides whether to normalize the input variables (True) or not (False).
- Would we like to pre-process the features to add model flexibility?
- What degree of regularization would we like to use in the model?
- How many model components would we like to use?

These are important choices that must be made *once the model class is selected*. These choices are often represented as *hyperparameters*, or parameters that must be set before the model is fit to data. This is done when you choose hyperparameters by passing values at model instantiation. For instance, you can specify you would like to fit the intercept using the fit_intercept hyperparameter:

```
In[3]: model = LinearRegression(fit_intercept=True)
Model

Out[3]: LinearRegression()
```

Notice that when the model is instantiated, the only action is the storing of the hyperparameter values. In particular, the model has not yet been applied to any data

iii). Arrange data into a features matrix and target vector.

Scikit-Learn data representation, requires a two-dimensional features matrix and a one-dimensional target array. The target variable y is in the correct form (length-n_sample array), but you need to reshape data x into two-dimensional array i.e. make it a matrix of size [n_samples, n_features].

```
In[4]: X = x[:, np.newaxis]
X.shape
Out[4]: (40, 1)
```

iv). Fit the model to your data.

Apply the model to data. This can be done with the fit() method of the model as follows:

```
In[5]: model.fit(X, y)
Out[5]:
LinearRegression()
```

The fit() command causes a number of model-dependent internal computations to take place, and the results of these computations are stored in model specific attributes. In this linear model, we have the following:

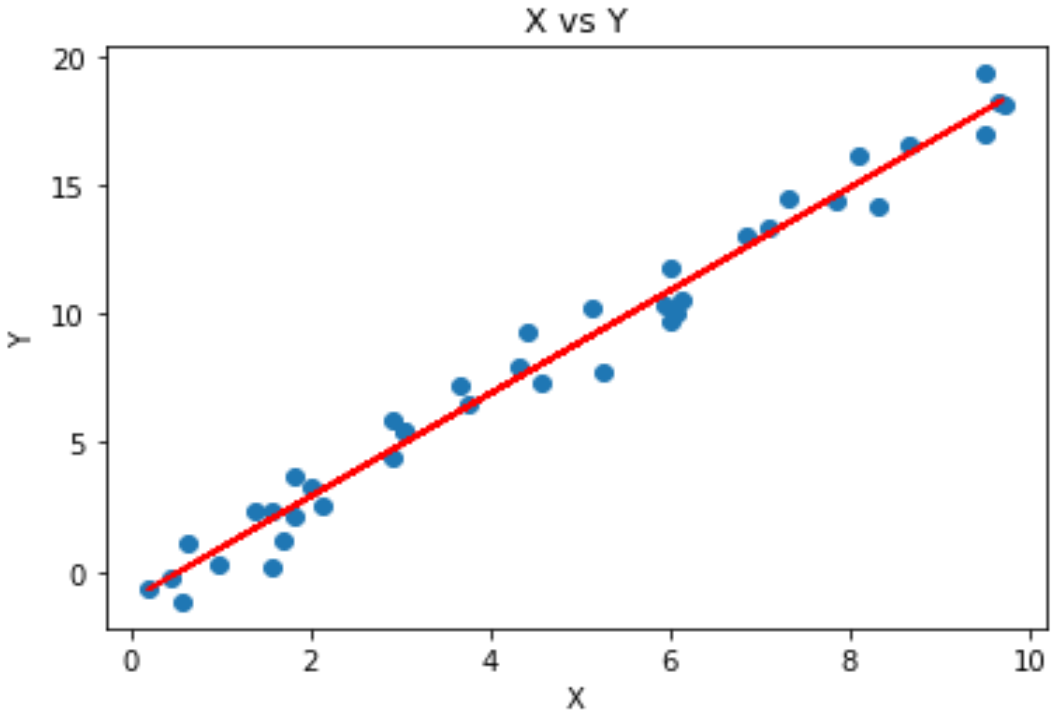
```
In[6]: model.coef_
Out[6]: array([2.0133901])

In[7]: model.intercept_
Out[7]: -1.139509001948376
```

These two parameters represent the slope and intercept of the simple linear fit to the data.

Based on the two parameters, you can visualize the results by plotting the best fit line against the raw data as shown below:

```
In[8]: plt.title("X vs Y")
plt.scatter(x, y);
plt.plot(x, (model.coef_*x)+(model.intercept_),color='red')
plt.xlabel("X")
plt.ylabel("Y")
```

```
Out[8]:
```



v). Predict labels for unknown data.

Once the model is trained, the main task of supervised machine learning is to evaluate it based on what it says about new data that was not part of the training set. In Scikit-Learn, this can done by using the predict() method. In this example, the "new data" will be a grid of *x* values, and we will need to know what *y* values the model predicts:

```
In[9]: xfit = np.linspace(-1, 11, 40)
```

Change the *x* values into a [n_samples, n_features] features matrix format and which can be feed to the model as shown below:

```
In[10]: Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

vi). Check the results of model fitting to know whether the model is satisfactory

Typically one evaluates the efficacy of the model by comparing its results to some known baseline.

Model Performance (Optimization)

The Goodness of fit determines how the line of regression fits the set of observations. This is achieved by using *R-squared method* (R^2), which is a statistical method that determines the goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance. It measures the strength of the relationship between the dependent and independent

variables on a scale of 0-100%. The high value of R-square the less difference between the predicted values and actual values and hence represents a good model. It is also called a *coefficient of determination***,** or *coefficient of multiple determination***.** It is computed by using the score()function.

```
In[11]:
r_sq = model.score(X, y)
print('coefficient of determination:', r_sq)

Out[11]:
coefficient of determination: 0.9780949186629684
```

Cost function

The cost function is used to find the accuracy of the *mapping function (hypothesis function)*. For Linear Regression, the *Mean Squared Error (MSE)* cost function is used. MSE is calculated as follows:

$$MSE = 1\frac{1}{N}\sum_{i=1}^{n}(y_i - (a_1x_i + a_0))^2$$

Where,

n=Total number of observation/examples

y_i = Actual value

$(a_1x_i+a_0)$= Predicted value.

In Python, MSE is computed as follows:

```
In[12]:
from sklearn import metrics

print('Mean Squared Error:',
metrics.mean_squared_error(y,(model.coef_*x)+(model.intercept_)))

Out[12]:
 Mean Squared Error: 0.7738965173505654
```

Multiple Linear Regressions (MLR)

MLR models linear relationship between a single dependent continuous variable and more than one independent variable E.g. Prediction of CO_2 emission based on engine size and number of cylinders in a car.

MLR equation

The target variable(Y) is a linear combination of multiple predictor variables x_1, x_2, x_3, ...,x_n. Since it is an enhancement of Simple Linear Regression, the same kind of equation is applied for the multiple linear regression as shown below:

$$Y= b_0+b_1x_1+ b_2x_2+ b_3x_3+..... b_nx_n$$

Where,

Y= Output/Response variable

b_0, b_1, b_2, b_3 , b_n....= Coefficients of the model.

x_1, x_2, x_3, x_4...= Various Independent/feature variable

If there are just two independent variables, the estimated regression function is $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2$. It represents a regression plane in a three-dimensional space. The goal of regression is to determine the values of the weights b_0, b_1, and b_2 such that this plane is as close as possible to the actual responses

Multiple Linear Regression with scikit-learn

Multiple linear regression is implemented using the same steps as simple regression.

i). Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
ii). Choose model hyperparameters by instantiating the class with desired values.
iii). Fit the model to your data by calling the *fit()* method of the model instance.
iv). Apply the model to new data
v). Check the results of model fitting to know whether the model is satisfactory

Example

Consider the following dataset

x_1	x_2	Y
0	1	4
5	1	5
15	2	20
25	5	14
35	11	32
45	15	22
55	34	38
60	35	43

i). Choose a class of model by importing the appropriate estimator class from Scikit-Learn.
import `numpy` and `sklearn.linear_model.LinearRegression` and provide inputs and output:

```
In[1]: import numpy as np
from sklearn.linear_model import LinearRegression
x = [[0, 1], [5, 1], [15, 2], [25, 5], [35, 11], [45, 15], [55, 34],
[60, 35]]
y = [4, 5, 20, 14, 32, 22, 38, 43]
x, y = np.array(x), np.array(y)
print(x)
print(y)

Out[1]:
[[ 0  1]
 [ 5  1]
 [15  2]
 [25  5]
 [35 11]
 [45 15]
 [55 34]
 [60 35]]
[ 4  5 20 14 32 22 38 43]
```

x is a two-dimensional array with two columns, while y is a one-dimensional array.

ii). Choose model hyperparameters by instantiating the class with desired values.

```
In[2]: model = LinearRegression()
```

iii). Fit the model to your data by calling the *fit()* method of the model instance.

```
In[3]: model.fit(x, y)
```

You can combine steps (ii) and (iii) as follows

```
In[4]: model = LinearRegression().fit(x,y)
```

The result of this statement is the variable model referring to the object of type `LinearRegression.` It represents the regression model fitted with existing data.

iv). Get results
You can obtain the properties of the model the same way as in the case of simple linear regression:

```
In[5]: print('intercept:', model.intercept_)

Out[5]: intercept: 5.5225792751981935

In[6]: print('slope:', model.coef_)

Out[6]: slope: [0.44706965 0.25502548]
```

`intercept_` holds the bias b_0, while `coef_` is an array containing b_1 and b_2 respectively. Notice that the intercept is approximately 5.52, and this is the value of the predicted response when $x_1 = x_2 = 0$. The increase of x_1 by 1 yields the rise of the predicted response by 0.45. Similarly, when x_2 grows by 1, the response rises by 0.26.

v). Apply the model to new data:
Predictions work the same way as simple linear regression as shown below:

```
In[7]: y_pred = model.predict(x)
       print('predicted response:', y_pred, sep='\n')

Out[7]:
predicted response:
[5.77760476  8.012953   12.73867497 17.9744479  23.97529728
 29.4660957 38.78227633 41.27265006]
```

This model can be applied to new data as well. First generate the data

```
In[8]: x_new = np.arange(10).reshape((-1, 2))
       print(x_new)
Out[8]:
       [[0 1]
        [2 3]
        [4 5]
        [6 7]
        [8 9]]
```

The following is prediction using the new data.

```
In[9]: y_new = model.predict(x_new)
       print(y_new)
Out[9]:
[ 5.77760476  7.18179502  8.58598528  9.99017554 11.3943658 ]
```

vi). Check the results of model fitting to know whether the model is satisfactory
The value of R^2 is obtained using `.score()` as follows:

```
In[10]: r_sq = model.score(x, y)
     print('coefficient of determination:', r_sq)

Out[10]:
coefficient of determination: 0.8615939258756776
```