Linear Regression

Python Packages for Linear Regression ¶

In [1]:
```
import numpy as np
from sklearn.linear_model import LinearRegression
```

In [2]:
```
x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))
y = np.array([5, 20, 14, 32, 22, 38])
```

In [3]:
```
x
```

Out[3]:
```
array([[ 5],
       [15],
       [25],
       [35],
       [45],
       [55]])
```

In [4]:
```
y
```

Out[4]:
```
array([ 5, 20, 14, 32, 22, 38])
```

Create a model and fit it

In [7]:
```
model=LinearRegression()
```

This statement creates the variable model as an instance of LinearRegression. You can provide several optional parameters to LinearRegression:

fit_intercept is a Boolean that, if True, decides to calculate the intercept b_0 or, if False, considers it equal to zero. It defaults to True.

normalize is a Boolean that, if True, decides to normalize the input variables. It defaults to False, in which case it doesn't normalize the input variables.

copy_X is a Boolean that decides whether to copy (True) or overwrite the input variables (False). It's True by default.

n_jobs is either an integer or None. It represents the number of jobs used in parallel computation. It defaults to None, which usually means one job. -1 means to use all available processors.

Your model as defined above uses the default values of all parameters.

It’s time to start using the model. First, you need to call .fit() on model:

```
In [6]: model.fit(x,y)
```

```
Out[6]: LinearRegression()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

With .fit(), you calculate the optimal values of the weights b_0 and b_1, using the existing input and output, x and y, as the arguments. In other words, .fit() fits the model. It returns self, which is the variable model itself. That’s why you can replace the last two statements with this one:

```
In [8]: model=LinearRegression().fit(x,y)
```

This statement does the same thing as the previous two. It’s just shorter.

Once you have your model fitted, you can get the results to check whether the model works satisfactorily and to interpret it.

You can obtain the coefficient of determination, R^2, with .score() called on model:

```
In [10]: r_square=model.score(x,y)
         print(f"coefficient of determination: {r_square}")
```

```
coefficient of determination: 0.715875613747954
```

When you’re applying .score(), the arguments are also the predictor x and response y, and the return value is R^2.

The attributes of model are .intercept_, which represents the coefficient b_0, and .coef_, which represents b_1:

```
In [12]: model_intercept=model.intercept_
         print(f"intercept : {model_intercept}")
```

```
intercept : 5.633333333333333
```

```
In [13]: model_coef=model.coef_
         print(f"Slope : {model_coef}")
```

```
Slope : [0.54]
```

The code above illustrates how to get b_0 and b_1. You can notice that .intercept_ is a scalar, while .coef_ is an array.

The value of b_0 is approximately 5.63. This illustrates that your model predicts the response 5.63 when x is zero. The value b_1 = 0.54 means that the predicted response rises by 0.54 when x is increased by one.

In [ ]: