**Simple Linear Regression With scikit-learn**

Let’s start with the simplest case, which is simple linear regression.

There are five basic steps when you’re implementing linear regression:

1. Import the packages and classes you need.
2. Provide data to work with and eventually do appropriate transformations.
3. Create a regression model and fit it with existing data.
4. Check the results of model fitting to know whether the model is satisfactory.
5. Apply the model for predictions.

These steps are more or less general for most of the regression approaches and implementations.

**Step 1: Import packages and classes**

The first step is to import the package numpy and the class LinearRegression from sklearn.linear\_model:

import numpy as np

from sklearn.linear\_model import LinearRegression

Now, you have all the functionalities you need to implement linear regression.

The fundamental data type of NumPy is the array type called numpy.ndarray. The rest of this article uses the term **array** to refer to instances of the type numpy.ndarray.

The class sklearn.linear\_model.LinearRegression will be used to perform linear and polynomial regression and make predictions accordingly.

**Step 2: Provide data**

The second step is defining data to work with. The inputs (regressors, 𝑥) and output (predictor, 𝑦) should be arrays (the instances of the class numpy.ndarray) or similar objects. This is the simplest way of providing data for regression:

x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))

y = np.array([5, 20, 14, 32, 22, 38])

Now, you have two arrays: the input x and output y. You should call .reshape() on x because this array is required to be **two-dimensional**, or to be more precise, to have **one column and as many rows as necessary**. That’s exactly what the argument (-1, 1) of .reshape() specifies.

This is how x and y look now:

>>>

>>> print(x)

[[ 5]

[15]

[25]

[35]

[45]

[55]]

>>> print(y)

[ 5 20 14 32 22 38]

As you can see, x has two dimensions, and x.shape is (6, 1), while y has a single dimension, and y.shape is (6,).

**Step 3: Create a model and fit it**

The next step is to create a linear regression model and fit it using the existing data.

Let’s create an instance of the class LinearRegression, which will represent the regression model:

model = LinearRegression()

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

* **fit\_intercept** is a Boolean (True by default) that decides whether to calculate the intercept 𝑏₀ (True) or consider it equal to zero (False).
* **normalize** is a Boolean (False by default) that decides whether to normalize the input variables (True) or not (False).
* **copy\_X** is a Boolean (True by default) that decides whether to copy (True) or overwrite the input variables (False).
* **n\_jobs** is an integer or None (default) and represents the number of jobs used in parallel computation. None usually means one job and -1 to use all processors.

This example uses the default values of all parameters.

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

model.fit(x, y)

With .fit(), you calculate the optimal values of the weights 𝑏₀ and 𝑏₁, 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:

model = LinearRegression().fit(x, y)

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

**Step 4: Get results**

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

You can obtain the coefficient of determination (𝑅²) with .score() called on model:

>>>

>>> r\_sq = model.score(x, y)

>>> print('coefficient of determination:', r\_sq)

coefficient of determination: 0.715875613747954

When you’re applying .score(), the arguments are also the predictor x and regressor y, and the return value is 𝑅².

The attributes of model are .intercept\_, which represents the coefficient, 𝑏₀ and .coef\_, which represents 𝑏₁:

>>>

>>> print('intercept:', model.intercept\_)

intercept: 5.633333333333329

>>> print('slope:', model.coef\_)

slope: [0.54]

The code above illustrates how to get 𝑏₀ and 𝑏₁. You can notice that .intercept\_ is a scalar, while .coef\_ is an array.

The value 𝑏₀ = 5.63 (approximately) illustrates that your model predicts the response 5.63 when 𝑥 is zero. The value 𝑏₁ = 0.54 means that the predicted response rises by 0.54 when 𝑥 is increased by one.

You should notice that you can provide y as a two-dimensional array as well. In this case, you’ll get a similar result. This is how it might look:

>>>

>>> new\_model = LinearRegression().fit(x, y.reshape((-1, 1)))

>>> print('intercept:', new\_model.intercept\_)

intercept: [5.63333333]

>>> print('slope:', new\_model.coef\_)

slope: [[0.54]]

As you can see, this example is very similar to the previous one, but in this case, .intercept\_ is a one-dimensional array with the single element 𝑏₀, and .coef\_ is a two-dimensional array with the single element 𝑏₁.

**Step 5: Predict response**

Once there is a satisfactory model, you can use it for predictions with either existing or new data.

To obtain the predicted response, use .predict():

>>>

>>> y\_pred = model.predict(x)

>>> print('predicted response:', y\_pred, sep='\n')

predicted response:

[ 8.33333333 13.73333333 19.13333333 24.53333333 29.93333333 35.33333333]

When applying .predict(), you pass the regressor as the argument and get the corresponding predicted response.

This is a nearly identical way to predict the response:

>>>

>>> y\_pred = model.intercept\_ + model.coef\_ \* x

>>> print('predicted response:', y\_pred, sep='\n')

predicted response:

[[ 8.33333333]

[13.73333333]

[19.13333333]

[24.53333333]

[29.93333333]

[35.33333333]]

In this case, you multiply each element of x with model.coef\_ and add model.intercept\_ to the product.

The output here differs from the previous example only in dimensions. The predicted response is now a two-dimensional array, while in the previous case, it had one dimension.

If you reduce the number of dimensions of x to one, these two approaches will yield the same result. You can do this by replacing x with x.reshape(-1), x.flatten(), or x.ravel() when multiplying it with model.coef\_.

In practice, regression models are often applied for forecasts. This means that you can use fitted models to calculate the outputs based on some other, new inputs:

>>>

>>> x\_new = np.arange(5).reshape((-1, 1))

>>> print(x\_new)

[[0]

[1]

[2]

[3]

[4]]

>>> y\_new = model.predict(x\_new)

>>> print(y\_new)

[5.63333333 6.17333333 6.71333333 7.25333333 7.79333333]

Here .predict() is applied to the new regressor x\_new and yields the response y\_new. This example conveniently uses arange() from numpy to generate an array with the elements from 0 (inclusive) to 5 (exclusive), that is 0, 1, 2, 3, and 4.

You can find more information about LinearRegression on [the official documentation page](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html).

Multiple Linear Regression With scikit-learn

You can implement multiple linear regression following the same steps as you would for simple regression.

**Steps 1 and 2: Import packages and classes, and provide data**

First, you import numpy and sklearn.linear\_model.LinearRegression and provide known inputs and output:

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)

That’s a simple way to define the input x and output y. You can print x and y to see how they look now:

>>>

>>> print(x)

[[ 0 1]

[ 5 1]

[15 2]

[25 5]

[35 11]

[45 15]

[55 34]

[60 35]]

>>> print(y)

[ 4 5 20 14 32 22 38 43]

In multiple linear regression, x is a two-dimensional array with at least two columns, while y is usually a one-dimensional array. This is a simple example of multiple linear regression, and x has exactly two columns.

**Step 3: Create a model and fit it**

The next step is to create the regression model as an instance of LinearRegression and fit it with .fit():

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.

**Step 4: Get results**

You can obtain the properties of the model the same way as in the case of simple linear regression:

>>>

>>> r\_sq = model.score(x, y)

>>> print('coefficient of determination:', r\_sq)

coefficient of determination: 0.8615939258756776

>>> print('intercept:', model.intercept\_)

intercept: 5.52257927519819

>>> print('slope:', model.coef\_)

slope: [0.44706965 0.25502548]

You obtain the value of 𝑅² using .score() and the values of the estimators of regression coefficients with .intercept\_ and .coef\_. Again, .intercept\_ holds the bias 𝑏₀, while now .coef\_ is an array containing 𝑏₁ and 𝑏₂ respectively.

In this example, the intercept is approximately 5.52, and this is the value of the predicted response when 𝑥₁ = 𝑥₂ = 0. The increase of 𝑥₁ by 1 yields the rise of the predicted response by 0.45. Similarly, when 𝑥₂ grows by 1, the response rises by 0.26.

**Step 5: Predict response**

Predictions also work the same way as in the case of simple linear regression:

>>>

>>> y\_pred = model.predict(x)

>>> print('predicted response:', y\_pred, sep='\n')

predicted response:

[ 5.77760476 8.012953 12.73867497 17.9744479 23.97529728 29.4660957

38.78227633 41.27265006]

The predicted response is obtained with .predict(), which is very similar to the following:

>>>

>>> y\_pred = model.intercept\_ + np.sum(model.coef\_ \* x, axis=1)

>>> print('predicted response:', y\_pred, sep='\n')

predicted response:

[ 5.77760476 8.012953 12.73867497 17.9744479 23.97529728 29.4660957

38.78227633 41.27265006]

You can predict the output values by multiplying each column of the input with the appropriate weight, summing the results and adding the intercept to the sum.

You can apply this model to new data as well:

>>>

>>> x\_new = np.arange(10).reshape((-1, 2))

>>> print(x\_new)

[[0 1]

[2 3]

[4 5]

[6 7]

[8 9]]

>>> y\_new = model.predict(x\_new)

>>> print(y\_new)

[ 5.77760476 7.18179502 8.58598528 9.99017554 11.3943658 ]

That’s the prediction using a linear regression model.