# ML5-sLR-Exercise_E5-1a

August 29, 2020

## 1 # Simple Linear Regression (sLR) With scikit-learn (Example from lesson ML05)

Powered by: Dr. Hermann Völlinger, DHBW Stuttgart(Germany); August 2020

Following ideas from: "Linear Regression in Python" by Mirko Stojiljkovic, 28.4.2020 (see details: https://realpython.com/linear-regression-in-python/#what-is-regression)

The example is from Lecture: "ML_Concept&Algorithm" (WS2020); Chapter ML5, Exercise E5.1-a "Exam Results"

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.

## 2 Step 1: Import packages and classes

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

[1]:
```
print␣
 ↪("*****************************************************************************")
print ("*** The picture shows the 10 students s1...s10 as points in (x,y)-plane␣
 ↪*****")
print ("*** X-axis --> hours of effort for prep. of exam; Y-axis␣
 ↪-->score-points ****")
print␣
 ↪("*****************************************************************************")

from IPython.display import Image

Image('test.jpg')
```

```


# Step 1: Import packages and classes

import numpy as np
from sklearn.linear_model import LinearRegression

# import time module
import time
```

```
*****************************************************************************
*** The picture shows the 10 students s1…s10 as points in (x,y)-plane *****
*** X-axis --> hours of effort for prep. of exam; Y-axis -->score-points ****
*****************************************************************************
```

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.

## 3 Step 2: Provide data

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

[2]:

```
# Step 2: Provide data

x = np.array([ 7, 3, 5, 3, 8, 7, 10, 3, 5, 3]).reshape((-1, 1))
y = np.array([41,27,35,26,48,45,46, 27,29,19])
```

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.

[3]:

```
print ("This is how x and y look now:")
print("x=",x)
print("y=",y)
```

```
This is how x and y look now:
x= [[ 7]
 [ 3]
 [ 5]
 [ 3]
 [ 8]
 [ 7]
 [10]
 [ 3]
 [ 5]
 [ 3]]
y= [41 27 35 26 48 45 46 27 29 19]
```

As you can see, x has two dimensions, and x.shape is (10, 1), while y has only a single dimension, and y.shape is (10,).

## 4 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:

```
[4]: 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 (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:

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

```
[5]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

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

```
[6]: # model = LinearRegression().fit(x, y)
```

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

## 5 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 ( $^2$) with .score() called on model:

[7]:

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

```
coefficient of determination: 0.8544449495100991
```

When you're applying .score(), the arguments are also the predictor x and regressor y, and the return value is $^2$.

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

[8]:

```
print('intercept:', model.intercept_)
print('slope:', model.coef_)
```

```
intercept: 14.11702127659574
slope: [3.73758865]
```

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

The value = 14.1170 (approximately) illustrates that your model predicts the response 14.1170 when is zero. The value = 3.7376 means that the predicted response rises by 3.7376 when 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:

[9]:

```
new_model = LinearRegression().fit(x, y.reshape((-1, 1)))
print('intercept:', new_model.intercept_)
print('slope:', new_model.coef_)
```

```
intercept: [14.11702128]
slope: [[3.73758865]]
```

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 , and .coef_ is a two-dimensional array with the single element .

# 6 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():

[10]:

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

```
predicted response:
[40.28014184 25.32978723 32.80496454 25.32978723 44.0177305  40.28014184
 51.4929078  25.32978723 32.80496454 25.32978723]
```

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:

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 twodimensional 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) y_new = model.predict(x_new) print(y_new)

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.

[12]:

```python
# print current date and time
print("****** current date and time *******")
print("date and time:",time.strftime("%d.%m.%Y %H:%M:%S"))
print ("end")
```

```
****** current date and time *******
date and time: 29.08.2020 22:28:21
end
```