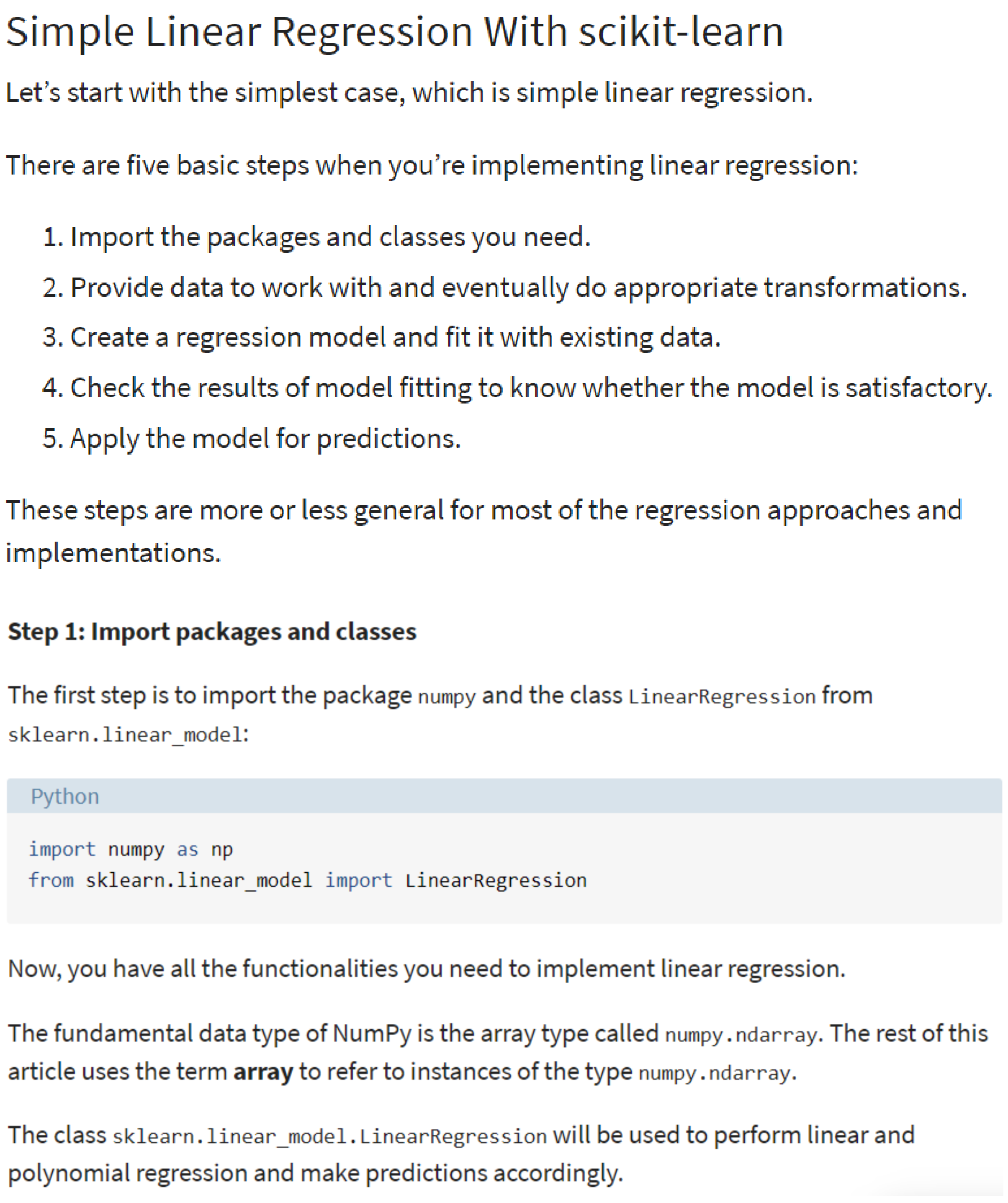
PYTHON CODE FOR ML PROBLEMS

URL: https://realpython.com/linear-regression-in-python/



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`:

Python

```python
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:

Python

```python
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:

Python >>>

```
>>> print(x)
[[ 5]
 [15]
 [25]
 [35]
 [45]
 [55]]
>>> print(y)
[ 5 20 14 32 22 38]
```

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:

Python

```python
model = LinearRegression()
```

This statement creates the variable `model` as the instance of `LinearRegression`. You can provide several optional parameters to `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 b_0 (`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`:

Python

```python
model.fit(x, y)
```

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:

Python

```python
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 (R^2) with `.score()` called on `model`:

Python >>>

```
>>> 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 R^2.

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

Python >>>

```
>>> print('intercept:', model.intercept_)
intercept: 5.633333333333329
>>> print('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 b_0 = 5.63 (approximately) 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.

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:

Python >>>

```
>>> 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 b_0, and `.coef_` is a two-dimensional array with the single element b_1.

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()`:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

```python
>>> 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`.

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:

```python
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:

Python >>>

```
>>> 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()`:

Python

```
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:

```
Python                                                              >>>

>>> 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 R^2 using `.score()` and the values of the estimators of regression coefficients with `.intercept_` and `.coef_`. Again, `.intercept_` holds the bias b_0, while now `.coef_` is an array containing b_1 and b_2 respectively.

In this example, 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.

Step 5: Predict response

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

Python >>>

```
>>> 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:

Python >>>

```
>>> 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 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:

Python >>>

```
>>> 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.

Polynomial Regression With scikit-learn

Implementing polynomial regression with scikit-learn is very similar to linear regression. There is only one extra step: you need to transform the array of inputs to include non-linear terms such as x^2.

Step 1: Import packages and classes

In addition to `numpy` and `sklearn.linear_model.LinearRegression`, you should also import the class `PolynomialFeatures` from `sklearn.preprocessing`:

Python

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
```

The import is now done, and you have everything you need to work with.

Step 2a: Provide data

This step defines the input and output and is the same as in the case of linear regression:

Python

```
x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))
y = np.array([15, 11, 2, 8, 25, 32])
```

Now you have the input and output in a suitable format. Keep in mind that you need the input to be a **two-dimensional array**. That's why `.reshape()` is used.

Step 2b: Transform input data

This is the **new step** you need to implement for polynomial regression!

As you've seen earlier, you need to include x^2 (and perhaps other terms) as additional features when implementing polynomial regression. For that reason, you should transform the input array `x` to contain the additional column(s) with the values of x^2 (and eventually more features).

It's possible to transform the input array in several ways (like using `insert()` from `numpy`), but the class `PolynomialFeatures` is very convenient for this purpose. Let's create an instance of this class:

```python
transformer = PolynomialFeatures(degree=2, include_bias=False)
```

The variable `transformer` refers to an instance of `PolynomialFeatures` which you can use to transform the input `x`.

You can provide several optional parameters to `PolynomialFeatures`:

- **`degree`** is an integer (`2` by default) that represents the degree of the polynomial regression function.
- **`interaction_only`** is a Boolean (`False` by default) that decides whether to include only interaction features (`True`) or all features (`False`).
- **`include_bias`** is a Boolean (`True` by default) that decides whether to include the bias (intercept) column of ones (`True`) or not (`False`).

This example uses the default values of all parameters, but you'll sometimes want to experiment with the degree of the function, and it can be beneficial to provide this argument anyway.

Before applying `transformer`, you need to fit it with `.fit()`:

Python

```python
transformer.fit(x)
```

Once `transformer` is fitted, it's ready to create a new, modified input. You apply `.transform()` to do that:

Python

```python
x_ = transformer.transform(x)
```

That's the transformation of the input array with `.transform()`. It takes the input array as the argument and returns the modified array.

You can also use `.fit_transform()` to replace the three previous statements with only one:

Python

```python
x_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(x)
```

That's fitting and transforming the input array in one statement with `.fit_transform()`. It also takes the input array and effectively does the same thing as `.fit()` and `.transform()` called in that order. It also returns the modified array. This is how the new input array looks:

Python

```
>>> print(x_)
[[   5.   25.]
 [  15.  225.]
 [  25.  625.]
 [  35. 1225.]
 [  45. 2025.]
 [  55. 3025.]]
```

The modified input array contains two columns: one with the original inputs and the other with their squares.

You can find more information about `PolynomialFeatures` on the official documentation page.

Step 3: Create a model and fit it

This step is also the same as in the case of linear regression. You create and fit the model:

Python

```
model = LinearRegression().fit(x_, y)
```

The regression model is now created and fitted. It's ready for application.

You should keep in mind that the first argument of `.fit()` is the *modified input array* `x_` and not the original `x`.

Step 4: Get results

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

Python >>>

```
>>> r_sq = model.score(x_, y)
>>> print('coefficient of determination:', r_sq)
coefficient of determination: 0.8908516262498564
>>> print('intercept:', model.intercept_)
intercept: 21.372321428571425
>>> print('coefficients:', model.coef_)
coefficients: [-1.32357143  0.02839286]
```

Again, `.score()` returns R^2. Its first argument is also the modified input `x_`, not `x`. The values of the weights are associated to `.intercept_` and `.coef_`: `.intercept_` represents b_0, while `.coef_` references the array that contains b_1 and b_2 respectively.

You can obtain a very similar result with different transformation and regression arguments:

Python

```
x_ = PolynomialFeatures(degree=2, include_bias=True).fit_transform(x)
```

If you call `PolynomialFeatures` with the default parameter `include_bias=True` (or if you just omit it), you'll obtain the new input array `x_` with the additional leftmost column containing only ones. This column corresponds to the intercept. This is how the modified input array looks in this case:

Python >>>

```
>>> print(x_)
[[1.000e+00 5.000e+00 2.500e+01]
 [1.000e+00 1.500e+01 2.250e+02]
 [1.000e+00 2.500e+01 6.250e+02]
 [1.000e+00 3.500e+01 1.225e+03]
 [1.000e+00 4.500e+01 2.025e+03]
 [1.000e+00 5.500e+01 3.025e+03]]
```

The first column of `x_` contains ones, the second has the values of `x`, while the third holds the squares of `x`.

The intercept is already included with the leftmost column of ones, and you don't need to include it again when creating the instance of `LinearRegression`. Thus, you can provide `fit_intercept=False`. This is how the next statement looks:

Python

```
model = LinearRegression(fit_intercept=False).fit(x_, y)
```

The variable `model` again corresponds to the new input array `x_`. Therefore `x_` should be passed as the first argument instead of `x`.

This approach yields the following results, which are similar to the previous case:

Python >>>

```
>>> r_sq = model.score(x_, y)
>>> print('coefficient of determination:', r_sq)
coefficient of determination: 0.8908516262498565
>>> print('intercept:', model.intercept_)
intercept: 0.0
>>> print('coefficients:', model.coef_)
coefficients: [21.37232143 -1.32357143  0.02839286]
```

You see that now `.intercept_` is zero, but `.coef_` actually contains b_0 as its first element. Everything else is the same.

Step 5: Predict response

If you want to get the predicted response, just use `.predict()`, but remember that the argument should be the modified input `x_` instead of the old `x`:

Python

```
>>> y_pred = model.predict(x_)
>>> print('predicted response:', y_pred, sep='\n')
predicted response:
[15.46428571  7.90714286  6.02857143  9.82857143 19.30714286 34.46428571]
```

As you can see, the prediction works almost the same way as in the case of linear regression. It just requires the modified input instead of the original.

You can apply the identical procedure if you have **several input variables**. You'll have an input array with more than one column, but everything else is the same. Here is an example:

Python

```
# Step 1: Import packages
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures

# Step 2a: Provide data
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)

# Step 2b: Transform input data
x_ = PolynomialFeatures(degree=2, include_bias=False).fit_transform(x)
```

```
# Step 3: Create a model and fit it
model = LinearRegression().fit(x_, y)

# Step 4: Get results
r_sq = model.score(x_, y)
intercept, coefficients = model.intercept_, model.coef_

# Step 5: Predict
y_pred = model.predict(x_)
```

This regression example yields the following results and predictions:

Python

```
>>> print('coefficient of determination:', r_sq)
coefficient of determination: 0.9453701449127822
>>> print('intercept:', intercept)
intercept: 0.8430556452395734
>>> print('coefficients:', coefficients, sep='\n')
coefficients:
[ 2.44828275  0.16160353 -0.15259677  0.47928683 -0.4641851 ]
>>> print('predicted response:', y_pred, sep='\n')
predicted response:
[ 0.54047408 11.36340283 16.07809622 15.79139    29.73858619 23.50834636
 39.05631386 41.92339046]
```

In this case, there are six regression coefficients (including the intercept), as shown in the estimated regression function $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2 + b_3x_1^2 + b_4x_1x_2 + b_5x_2^2$.

You can also notice that polynomial regression yielded a higher coefficient of determination than multiple linear regression for the same problem. At first, you could think that obtaining such a large R^2 is an excellent result. It might be.

However, in real-world situations, having a complex model and R^2 very close to 1 might also be a sign of overfitting. To check the performance of a model, you should test it with new data, that is with observations not used to fit (train) the model.

Advanced Linear Regression With `statsmodels`

You can implement linear regression in Python relatively easily by using the package `statsmodels` as well. Typically, this is desirable when there is a need for more detailed results.

The procedure is similar to that of scikit-learn.

Step 1: Import packages

First you need to do some imports. In addition to `numpy`, you need to import `statsmodels.api`:

```python
import numpy as np
import statsmodels.api as sm
```

Now you have the packages you need.

Step 2: Provide data and transform inputs

You can provide the inputs and outputs the same way as you did when you were using scikit-learn:

```python
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)
```

The input and output arrays are created, but the job is not done yet.

You need to add the column of ones to the inputs if you want `statsmodels` to calculate the intercept b_0. It doesn't takes b_0 into account by default. This is just one function call:

```python
x = sm.add_constant(x)
```

That's how you add the column of ones to `x` with `add_constant()`. It takes the input array `x` as an argument and returns a new array with the column of ones inserted at the beginning. This is how `x` and `y` look now:

Python

```
>>> print(x)
[[ 1.  0.  1.]
 [ 1.  5.  1.]
 [ 1. 15.  2.]
 [ 1. 25.  5.]
 [ 1. 35. 11.]
 [ 1. 45. 15.]
 [ 1. 55. 34.]
 [ 1. 60. 35.]]
>>> print(y)
[ 4  5 20 14 32 22 38 43]
```

You can see that the modified `x` has three columns: the first column of ones (corresponding to b_0 and replacing the intercept) as well as two columns of the original features.

Step 3: Create a model and fit it

The regression model based on ordinary least squares is an instance of the class `statsmodels.regression.linear_model.OLS`. This is how you can obtain one:

Python

```
model = sm.OLS(y, x)
```

You should be careful here! Please, notice that the first argument is the output, followed with the input. There are several more optional parameters.

To find more information about this class, please visit the official documentation page.

Once your model is created, you can apply `.fit()` on it:

Python

```
results = model.fit()
```

By calling `.fit()`, you obtain the variable `results`, which is an instance of the class `statsmodels.regression.linear_model.RegressionResultsWrapper`. This object holds a lot of information about the regression model.

Step 4: Get results

The variable `results` refers to the object that contains detailed information about the results of linear regression. Explaining them is far beyond the scope of this article, but you'll learn here how to extract them.

You can call `.summary()` to get the table with the results of linear regression:

Python >>>

```
>>> print(results.summary())
OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.862
Model:                            OLS   Adj. R-squared:                  0.806
Method:                 Least Squares   F-statistic:                     15.56
Date:                Sun, 17 Feb 2019   Prob (F-statistic):            0.00713
Time:                        19:15:07   Log-Likelihood:                -24.316
No. Observations:                   8   AIC:                             54.63
Df Residuals:                       5   BIC:                             54.87
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.5226      4.431      1.246      0.268      -5.867      16.912
x1             0.4471      0.285      1.567      0.178      -0.286       1.180
x2             0.2550      0.453      0.563      0.598      -0.910       1.420
==============================================================================
Omnibus:                        0.561   Durbin-Watson:                   3.268
Prob(Omnibus):                  0.755   Jarque-Bera (JB):                0.534
Skew:                           0.380   Prob(JB):                        0.766
Kurtosis:                       1.987   Cond. No.                         80.1
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly spec
```

This table is very comprehensive. You can find many statistical values associated with linear regression including R^2, b_0, b_1, and b_2.

In this particular case, you might obtain the warning related to `kurtosistest`. This is due to the small number of observations provided.

You can extract any of the values from the table above. Here's an example:

Python >>>

```
>>> print('coefficient of determination:', results.rsquared)
coefficient of determination: 0.8615939258756777
>>> print('adjusted coefficient of determination:', results.rsquared_adj)
adjusted coefficient of determination: 0.8062314962259488
>>> print('regression coefficients:', results.params)
regression coefficients: [5.52257928 0.44706965 0.25502548]
```

That's how you obtain some of the results of linear regression: