**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 that you need to implement linear regression.

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

You’ll use the class sklearn.linear\_model.LinearRegression 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 (response, 𝑦) should be arrays 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 the output, y. You should call .reshape() on x because this array must be **two-dimensional**, or more precisely, it must 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:

>>>

>>> x

array([[ 5],

[15],

[25],

[35],

[45],

[55]])

>>> y

array([ 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.

Create an instance of the class LinearRegression, which will represent the regression model:

>>>

>>> model = LinearRegression()

This statement creates the [variable](https://realpython.com/python-variables/) model as an instance of LinearRegression. You can provide several optional parameters to LinearRegression:

* **fit\_intercept** is a [Boolean](https://realpython.com/python-boolean/) that, if True, decides to calculate the intercept 𝑏₀ 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:

>>>

>>> model.fit(x, y)

LinearRegression()

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 to interpret it.

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

>>>

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

>>> print(f"coefficient of determination: {r\_sq}")

coefficient of determination: 0.7158756137479542

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

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

>>>

>>> print(f"intercept: {model.intercept\_}")

intercept: 5.633333333333329

>>> print(f"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.

**Note:** In scikit-learn, by [convention](https://scikit-learn.org/stable/developers/develop.html#estimated-attributes), a trailing underscore indicates that an attribute is estimated. In this example, .intercept\_ and .coef\_ are estimated values.

The value of 𝑏₀ is approximately 5.63. This 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’ll 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(f"intercept: {new\_model.intercept\_}")

intercept: [5.63333333]

>>> print(f"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 you have a satisfactory model, then 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(f"predicted response:\n{y\_pred}")

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(f"predicted response:\n{y\_pred}")

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, then 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 new inputs:

>>>

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

>>> x\_new

array([[0],

[1],

[2],

[3],

[4]])

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

>>> y\_new

array([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()](https://realpython.com/how-to-use-numpy-arange/) from numpy to generate an array with the elements from 0, inclusive, up to but excluding 5—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. The main difference is that your x array will now have two or more columns.

**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:

>>>

>>> x

array([[ 0, 1],

[ 5, 1],

[15, 2],

[25, 5],

[35, 11],

[45, 15],

[55, 34],

[60, 35]])

>>> y

array([ 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(f"coefficient of determination: {r\_sq}")

coefficient of determination: 0.8615939258756776

>>> print(f"intercept: {model.intercept\_}")

intercept: 5.52257927519819

>>> print(f"coefficients: {model.coef\_}")

coefficients: [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.

In this example, the intercept is approximately 5.52, and this is the value of the predicted response when 𝑥₁ = 𝑥₂ = 0. An increase of 𝑥₁ by 1 yields a 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(f"predicted response:\n{y\_pred}")

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 equivalent to the following:

>>>

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

>>> print(f"predicted response:\n{y\_pred}")

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))

>>> x\_new

array([[0, 1],

[2, 3],

[4, 5],

[6, 7],

[8, 9]])

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

>>> y\_new

array([ 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’s only one extra step: you need to transform the array of inputs to include nonlinear terms such as 𝑥².

**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:

>>>

>>> 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:

>>>

>>> 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** that you need to implement for polynomial regression!

As you learned earlier, you need to include 𝑥²—and perhaps other terms—as additional features when implementing polynomial regression. For that reason, you should transform the input array x to contain any additional columns with the values of 𝑥², 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. Go ahead and create an instance of this class:

>>>

>>> transformer = PolynomialFeatures(degree=2, include\_bias=False)

The variable transformer refers to an instance of PolynomialFeatures that 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, or intercept, column of 1 values (True) or not (False).

This example uses the default values of all parameters except include\_bias. You’ll sometimes want to experiment with the degree of the function, and it can be beneficial for readability to provide this argument anyway.

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

>>>

>>> transformer.fit(x)

PolynomialFeatures(include\_bias=False)

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

>>>

>>> 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:

>>>

>>> x\_ = PolynomialFeatures(degree=2, include\_bias=False).fit\_transform(x)

With .fit\_transform(), you’re fitting and transforming the input array in one statement. This method 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:

>>>

>>> x\_

array([[ 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.

**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:

>>>

>>> 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:

>>>

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

>>> print(f"coefficient of determination: {r\_sq}")

coefficient of determination: 0.8908516262498563

>>> print(f"intercept: {model.intercept\_}")

intercept: 21.372321428571436

>>> print(f"coefficients: {model.coef\_}")

coefficients: [-1.32357143 0.02839286]

Again, .score() returns 𝑅². Its first argument is also the modified input x\_, not x. The values of the weights are associated to .intercept\_ and .coef\_. Here, .intercept\_ represents 𝑏₀, while .coef\_ references the array that contains 𝑏₁ and 𝑏₂.

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

>>>

>>> 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, then you’ll obtain the new input array x\_ with the additional leftmost column containing only 1 values. This column corresponds to the intercept. This is how the modified input array looks in this case:

>>>

>>> x\_

array([[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:

>>>

>>> 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:

>>>

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

>>> print(f"coefficient of determination: {r\_sq}")

coefficient of determination: 0.8908516262498564

>>> print(f"intercept: {model.intercept\_}")

intercept: 0.0

>>> print(f"coefficients: {model.coef\_}")

coefficients: [21.37232143 -1.32357143 0.02839286]

You see that now .intercept\_ is zero, but .coef\_ actually contains 𝑏₀ 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:

>>>

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

>>> print(f"predicted response:\n{y\_pred}")

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 an identical procedure if you have **several input variables**. You’ll have an input array with more than one column, but everything else will be the same. Here’s an example:

>>>

>>> # Step 1: Import packages and classes

>>> 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 response

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

This regression example yields the following results and predictions:

>>>

>>> print(f"coefficient of determination: {r\_sq}")

coefficient of determination: 0.9453701449127822

>>> print(f"intercept: {intercept}")

intercept: 0.8430556452395876

>>> print(f"coefficients:\n{coefficients}")

coefficients:

[ 2.44828275 0.16160353 -0.15259677 0.47928683 -0.4641851 ]

>>> print(f"predicted response:\n{y\_pred}")

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(𝑥₁, 𝑥₂) = 𝑏₀ + 𝑏₁𝑥₁ + 𝑏₂𝑥₂ + 𝑏₃𝑥₁² + 𝑏₄𝑥₁𝑥₂ + 𝑏₅𝑥₂².

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 𝑅² is an excellent result. It might be.

However, in real-world situations, having a complex model and 𝑅² very close to one 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, or train, the model. To learn how to split your dataset into the training and test subsets,

**Advanced Linear Regression With statsmodels**

You can implement linear regression in Python by using the package statsmodels as well. Typically, this is desirable when you need 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:

>>>

>>> import numpy as np

>>> import statsmodels.api as sm

Now you have the packages that 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:

>>>

>>> 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 isn’t done yet.

You need to add the column of ones to the inputs if you want statsmodels to calculate the intercept 𝑏₀. It doesn’t take 𝑏₀ into account by default. This is just one function call:

>>>

>>> 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:

>>>

>>> x

array([[ 1., 0., 1.],

[ 1., 5., 1.],

[ 1., 15., 2.],

[ 1., 25., 5.],

[ 1., 35., 11.],

[ 1., 45., 15.],

[ 1., 55., 34.],

[ 1., 60., 35.]])

>>> y

array([ 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 𝑏₀ 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:

>>>

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

You should be careful here! Notice that the first argument is the output, followed by the input. This is the opposite order of the corresponding scikit-learn functions.

There are several more optional parameters.

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

>>>

>>> 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 these results is far beyond the scope of this tutorial, but you’ll learn here how to extract them.

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

>>>

>>> 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: Thu, 12 May 2022 Prob (F-statistic): 0.00713

Time: 14:15:07 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 specified.

This table is very comprehensive. You can find many statistical values associated with linear regression, including 𝑅², 𝑏₀, 𝑏₁, and 𝑏₂.

In this particular case, you might obtain a warning saying kurtosistest only valid for n>=20. This is due to the small number of observations provided in the example.

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

>>>

>>> print(f"coefficient of determination: {results.rsquared}")

coefficient of determination: 0.8615939258756776

>>> print(f"adjusted coefficient of determination: {results.rsquared\_adj}")

adjusted coefficient of determination: 0.8062314962259487

>>> print(f"regression coefficients: {results.params}")

regression coefficients: [5.52257928 0.44706965 0.25502548]

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

1. **.rsquared** holds 𝑅².
2. **.rsquared\_adj** represents adjusted 𝑅²—that is, 𝑅² corrected according to the number of input features.
3. **.params** refers the array with 𝑏₀, 𝑏₁, and 𝑏₂.

You can also notice that these results are identical to those obtained with scikit-learn for the same problem.

To find more information about the results of linear regression, please visit [the official documentation page](https://www.statsmodels.org/stable/generated/statsmodels.regression.linear_model.RegressionResults.html).

**Step 5: Predict response**

You can obtain the predicted response on the input values used for creating the model using .fittedvalues or .predict() with the input array as the argument:

>>>

>>> print(f"predicted response:\n{results.fittedvalues}")

predicted response:

[ 5.77760476 8.012953 12.73867497 17.9744479 23.97529728 29.4660957

38.78227633 41.27265006]

>>> print(f"predicted response:\n{results.predict(x)}")

predicted response:

[ 5.77760476 8.012953 12.73867497 17.9744479 23.97529728 29.4660957

38.78227633 41.27265006]

This is the predicted response for known inputs. If you want predictions with new regressors, you can also apply .predict() with new data as the argument:

>>>

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

>>> x\_new

array([[1., 0., 1.],

[1., 2., 3.],

[1., 4., 5.],

[1., 6., 7.],

[1., 8., 9.]])

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

>>> y\_new

array([ 5.77760476, 7.18179502, 8.58598528, 9.99017554, 11.3943658 ])

You can notice that the predicted results are the same as those obtained with scikit-learn for the same problem.

**Beyond Linear Regression**

Linear regression is sometimes not appropriate, especially for nonlinear models of high complexity.

Fortunately, there are other regression techniques suitable for the cases where linear regression doesn’t work well. Some of them are support vector machines, decision trees, random forest, and neural networks.

There are numerous Python libraries for regression using these techniques. Most of them are free and open-source. That’s one of the reasons why Python is among the main programming languages for machine learning.

The package scikit-learn provides the means for using other regression techniques in a very similar way to what you’ve seen. It contains classes for [support vector machines](https://scikit-learn.org/stable/modules/svm.html), [decision trees](https://scikit-learn.org/stable/modules/tree.html), [random forest](https://scikit-learn.org/stable/modules/ensemble.html#forests-of-randomized-trees), and more, with the methods .fit(), .predict(), .score(), and so on.