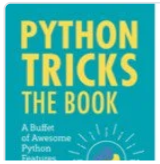
# Linear Regression in Python

by Mirko Stojiljković May 16, 2022 25 Comments data-science intermediate machine-learning

Mark as Completed

Tweet Share Email

## Table of Contents



Regression analysis is one of the most important fields in statistics and machine learning. There are many regression methods available. Linear regression is one of them.

## What Is Regression?

Regression searches for relationships among **variables**. For example, you can observe several employees of some company and try to understand how their salaries depend on their **features**, such as experience, education level, role, city of employment, and so on.

This is a regression problem where data related to each employee represents one **observation**. The presumption is that the experience, education, role, and city are the independent features, while the salary depends on them.

Similarly, you can try to establish the mathematical dependence of housing prices on area, number of bedrooms, distance to the city center, and so on.

Generally, in regression analysis, you consider some phenomenon of interest and have a number of observations. Each observation has two or more features. Following the assumption that at least one of the features depends on the others, you try to establish a relation among them.

In other words, you need to find a **function that maps some features or variables to others** sufficiently well.

The dependent features are called the **dependent variables**, **outputs**, or **responses**. The independent features are called the **independent variables**, **inputs**, **regressors**, or **predictors**.

Regression problems usually have one continuous and unbounded dependent variable. The inputs, however, can be continuous, discrete, or even categorical data such as gender, nationality, or brand.

It’s a common practice to denote the outputs with $y$ and the inputs with $x$. If there are two or more independent variables, then they can be represented as the vector $\mathbf{x} = (x_1, ..., x_r)$, where $r$ is the number of inputs.

## Regression Performance

The variation of actual responses $y_i$, $i = 1, …, n$, occurs partly due to the dependence on the predictors $\mathbf{x}_i$. However, there's also an additional inherent variance of the output.

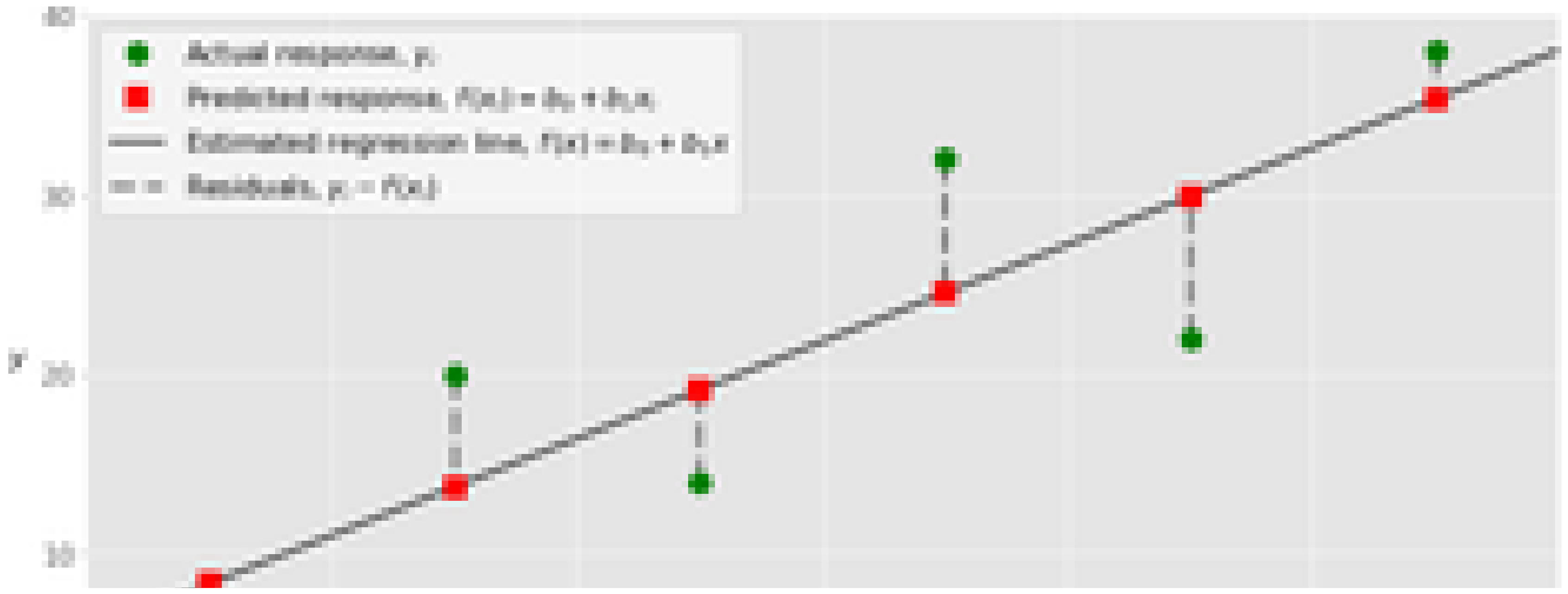
The **coefficient of determination**, denoted as $R^2$, tells you which amount of variation in $y$ can be explained by the dependence on $\mathbf{x}$, using the particular regression model. A larger $R^2$ indicates a better fit and means that the model can better explain the variation of the output with different inputs.

The value $R^2 = 1$ corresponds to SSR = 0. That's the **perfect fit**, since the values of predicted and actual responses fit completely to each other.

## Simple Linear Regression

Simple or single-variate linear regression is the simplest case of linear regression, as it has a single independent variable, $\mathbf{x} = x$.

The following figure illustrates simple linear regression:



You can regard polynomial regression as a generalized case of linear regression. You assume the polynomial dependence between the output and inputs and, consequently, the polynomial estimated regression function.

In other words, in addition to linear terms like $b_1x_1$, your regression function $f$ can include nonlinear terms such as $b_2x_1^2$, $b_3x_1^3$, or even $b_4x_1x_2$, $b_5x_1^2x_2$.

The simplest example of polynomial regression has a single independent variable, and the estimated regression function is a polynomial of degree two: $f(x) = b_0 + b_1x + b_2x^2$.

Now, remember that you want to calculate $b_0$, $b_1$, and $b_2$ to minimize SSR. These are your unknowns!

Keeping this in mind, compare the previous regression function with the function $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2$, used for linear regression. They look very similar and are both linear functions of the unknowns $b_0$, $b_1$, and $b_2$. This is why you can solve the **polynomial regression problem** as a **linear problem** with the term $x^2$ regarded as an input variable.

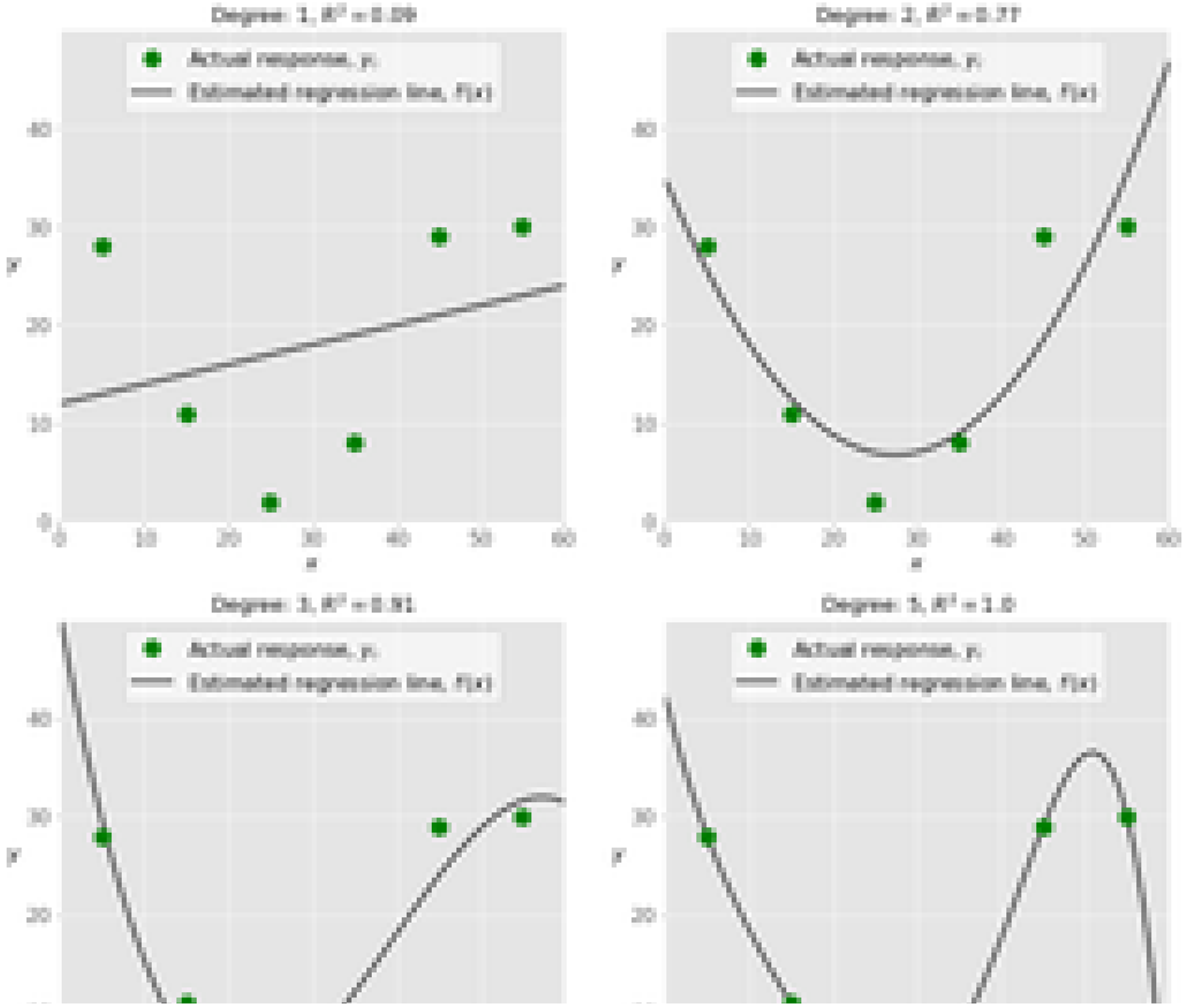
In the case of two variables and the polynomial of degree two, the regression function has this form: $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2 + b_3x_1^2 + b_4x_1x_2 + b_5x_2^2$.

The procedure for solving the problem is identical to the previous case. You apply linear regression for five inputs: $x_1$, $x_2$, $x_1^2$, $x_1x_2$, and $x_2^2$. As the result of regression, you get the values of six weights that minimize SSR: $b_0$, $b_1$, $b_2$, $b_3$, $b_4$, and $b_5$.

Of course, there are more general problems, but this should be enough to illustrate the point.

## Underfitting and Overfitting

One very important question that might arise when you're implementing polynomial regression is related to the choice of the **optimal degree** of the polynomial regression function.


Degree: 1, $R^2$ = 0.09
Degree: 2, $R^2$ = 0.77
Degree: 3, $R^2$ = 0.81
Degree: 5, $R^2$ = 1.0
Actual response, $y_i$
Estimated regression line, $f(x)$
y
x
0
10
20
30
40
50
60

If you're not familiar with NumPy, you can use the official NumPy User Guide and read NumPy Tutorial: Your First Steps Into Data Science in Python. In addition, Look Ma, No For-Loops: Array Programming With NumPy and Pure Python vs NumPy vs TensorFlow Performance Comparison can give you a good idea of the performance gains that you can achieve when applying NumPy.

The package **scikit-learn** is a widely used Python library for machine learning, built on top of NumPy and some other packages. It provides the means for preprocessing data, reducing dimensionality, implementing regression, classifying, clustering, and more. Like NumPy, scikit-learn is also open-source.

You can check the page Generalized Linear Models on the scikit-learn website to learn more about linear models and get deeper insight into how this package works.

If you want to implement linear regression and need functionality beyond the scope of scikit-learn, you should consider **statsmodels**. It's a powerful Python package for the estimation of statistical models, performing tests, and more. It's open-source as well.

You can find more information on statsmodels on its official website.

Now, to follow along with this tutorial, you should install all these packages into a virtual environment:

```
(venv) $ python -m pip install numpy scikit-learn statsmodels
```

This will install NumPy, scikit-learn, statsmodels, and their dependencies.

## Simple Linear Regression With scikit-learn

You'll 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 that you need.

```
>>> 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 `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`:

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.

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

### 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]
```

## 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.

```
>>> 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.

As you learned 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 any additional columns 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. 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)
>>> 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 $R^2$. 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 $b_0$, while `.coef_` references the array that contains $b_1$ and $b_2$.

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:

```
>>> # 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 ]
```

```
>>> 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 $b_0$. It doesn't take $b_0$ 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.],
>>> 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 $R^2$, $b_0$, $b_1$, and $b_2$.

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:

```
>>> 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, decision trees, random forest, and more, with the methods `.fit()`, `.predict()`, `.score()`, and so on.