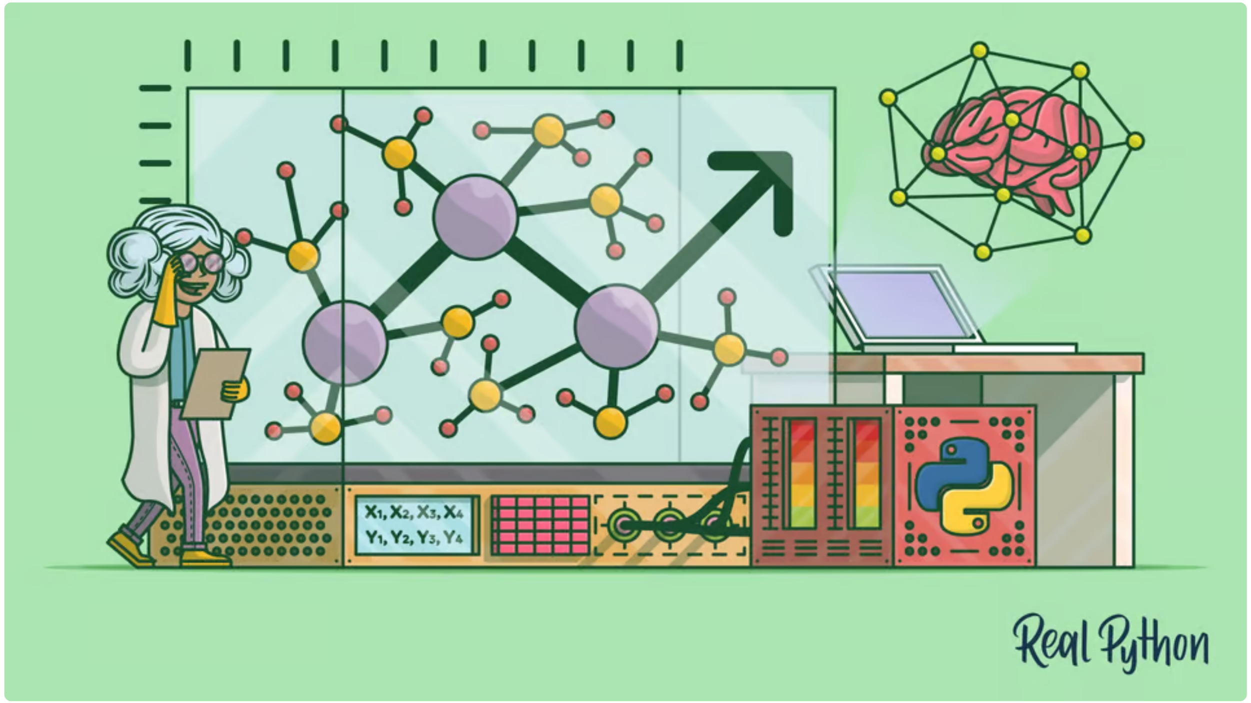


Linear Regression in Python

by Mirko Stojiljković May 16, 2022 26 Comments data-science intermediate machine-learning

Mark as Completed

Tweet Share Email

Table of Contents

Browse Topics
Guided Learning Paths
Basics Intermediate Advanced

api best-practices community databases data-science devops django docker flask front-end gamedev gui machine-learning projects python testing tools web-dev web-scraping

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Starting With Linear Regression in Python**

You're living in an era of large amounts of data, powerful computers, and artificial intelligence. This is just the beginning. Data science and machine learning are driving image recognition, development of autonomous vehicles, decisions in the financial and energy sectors, advances in medicine, the rise of social networks, and more. Linear regression is an important part of this.

Table of Contents

Mark as Completed

Linear regression is one of the fundamental statistical and machine learning techniques. Whether you want to do statistics, machine learning, or scientific computing, there's a good

chance that you'll need it. It's best to build a solid foundation first and then proceed toward more complex methods.

By the end of this article, you'll have learned:

- What linear regression **is**
- What linear regression is **used** for
- How linear regression **works**
- How to **implement** linear regression in Python, step by step

Free Bonus: Click here to get access to a free NumPy Resources Guide that points you to the best tutorials, videos, and books for improving your NumPy skills.

Take the Quiz: Test your knowledge with our interactive "Linear Regression in Python" quiz. Upon completion you will receive a score so you can track your learning progress over time:

Take the Quiz »

Regression

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.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com
PYTHONISTACAFE

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 $\mathbf{x} = (x_1, \ldots, x_r)$, where r is the number of inputs.

When Do You Need Regression?

Typically, you need regression to answer whether and how some phenomenon influences the other or how **several** variables are related. For example, you can use it to determine *if* and *to what extent* experience or gender impacts salaries.

Regression is also useful when you want to **forecast** a response using a new set of predictors. For example, you could try to predict electricity consumption of a household for the next hour given the outdoor temperature, time of day, and number of residents in that household.

Regression is used in many different fields, including economics, computer science, and the social sciences. Its importance rises every day with the availability of large amounts of data and increased awareness of the practical value of data.

Linear Regression

Linear regression is probably one of the most important and widely used regression techniques. It's among the simplest regression methods. One of its main advantages is the ease of interpreting results.

Problem Formulation

When implementing linear regression of some dependent variable y on the set of independent variables $\mathbf{x} = (x_1, \ldots, x_r)$, where r is the number of predictors, you assume a linear relationship between y and $\mathbf{x}$: $y = \beta_0 + \beta_1 x_1 + \cdots + \beta_r x_r + \varepsilon$. This equation is the **regression equation**. $\beta_0, \beta_1, \ldots, \beta_r$ are the **regression coefficients**, and ε is the **random error**.

Linear regression calculates the **estimators** of the regression coefficients or simply the

Linear regression calculates the **estimators** of the regression coefficients or simply the **predicted weights**, denoted with $b_0, b_1, \ldots, b_r$. These estimators define the **estimated**

regression function $f(\mathbf{x}) = b_0 + b_1x_1 + \cdots + b_rx_r$. This function should capture the dependencies between the inputs and output sufficiently well.

The **estimated** or **predicted response**, $f(\mathbf{x}_i)$, for each observation $i = 1, \ldots, n$, should be as close as possible to the corresponding **actual response** y_i. The differences $y_i - f(\mathbf{x}_i)$ for all observations $i = 1, \ldots, n$, are called the **residuals**. Regression is about determining the **best predicted weights**—that is, the weights corresponding to the smallest residuals.

To get the best weights, you usually **minimize the sum of squared residuals (SSR)** for all observations $i = 1, \ldots, n$: SSR = $\Sigma_i(y_i - f(\mathbf{x}_i))^2$. This approach is called the **method of ordinary least squares**.

Regression Performance

The variation of actual responses y_i, $i = 1, \ldots, n$, 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:

Example of simple linear regression

When implementing simple linear regression, you typically start with a given set of input-output (x-y) pairs. These pairs are your observations, shown as green circles in the figure. For example, the leftmost observation has the input x = 5 and the actual output, or response, y = 5. The next one has x = 15 and y = 20, and so on.

The estimated regression function, represented by the black line, has the equation $f(x) = b_0 + b_1x$. Your goal is to calculate the optimal values of the predicted weights b_0 and b_1 that minimize SSR and determine the estimated regression function.

The value of b_0, also called the **intercept**, shows the point where the estimated regression line crosses the y axis. It's the value of the estimated response $f(x)$ for $x = 0$. The value of b_1 determines the **slope** of the estimated regression line.

The predicted responses, shown as red squares, are the points on the regression line that correspond to the input values. For example, for the input $x = 5$, the predicted response is $f(5) = 8.33$, which the leftmost red square represents.

The vertical dashed grey lines represent the residuals, which can be calculated as $y_i - f(\mathbf{x}_i) = y_i - b_0 - b_1x_i$ for $i = 1, \ldots, n$. They're the distances between the green circles and red squares. When you implement linear regression, you're actually trying to minimize these distances and make the red squares as close to the predefined green circles as possible.

Multiple Linear Regression

Multiple or multivariate linear regression is a case of linear regression with two or more independent variables.

If there are just two independent variables, then the estimated regression function is $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2$. It represents a regression plane in a three-dimensional space. The goal of regression is to determine the values of the weights b_0, b_1, and b_2 such that this plane is as close as possible to the actual responses, while yielding the minimal SSR.

The case of more than two independent variables is similar, but more general. The estimated regression function is $f(x_1, \ldots, x_r) = b_0 + b_1x_1 + \cdots + b_rx_r$, and there are $r + 1$ weights to be determined when the number of inputs is r.

Polynomial 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

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.

There’s no straightforward rule for doing this. It depends on the case. You should, however, be aware of two problems that might follow the choice of the degree: **underfitting** and **overfitting**.

Underfitting occurs when a model can’t accurately capture the dependencies among data, usually as a consequence of its own simplicity. It often yields a low R^2 with known data and bad generalization capabilities when applied with new data.

Overfitting happens when a model learns both data dependencies and random fluctuations. In other words, a model learns the existing data too well. Complex models, which have many features or terms, are often prone to overfitting. When applied to known data, such models usually yield high R^2. However, they often don’t generalize well and have significantly lower R^2 when used with new data.

The next figure illustrates the underfitted, well-fitted, and overfitted models:

Example of underfitted, well-fitted and overfitted models

The top-left plot shows a linear regression line that has a low R^2. It might also be important

The top-left plot shows a linear regression line that has a low R^2. It might also be important that a straight line can't take into account the fact that the actual response increases as x moves away from twenty-five and toward zero. This is likely an example of underfitting.

The top-right plot illustrates polynomial regression with the degree equal to two. In this instance, this might be the optimal degree for modeling this data. The model has a value of R^2 that's satisfactory in many cases and shows trends nicely.

The bottom-left plot presents polynomial regression with the degree equal to three. The value of R^2 is higher than in the preceding cases. This model behaves better with known data than the previous ones. However, it shows some signs of overfitting, especially for the input values close to sixy, where the line starts decreasing, although the actual data doesn't show that.

Finally, on the bottom-right plot, you can see the perfect fit: six points and the polynomial line of the degree five (or higher) yield $R^2 = 1$. Each actual response equals its corresponding prediction.

In some situations, this might be exactly what you're looking for. In many cases, however, this is an overfitted model. It's likely to have poor behavior with unseen data, especially with the inputs larger than fifty.

For example, it assumes, without any evidence, that there's a significant drop in responses for x greater than fifty and that y reaches zero for x near sixty. Such behavior is the consequence of excessive effort to learn and fit the existing data.

There are a lot of resources where you can find more information about regression in general and linear regression in particular. The regression analysis page on Wikipedia, Wikipedia's linear regression entry, and Khan Academy's linear regression article are good starting points.

Python Packages for Linear Regression

It's time to start implementing linear regression in Python. To do this, you'll apply the proper packages and their functions and classes.

NumPy is a fundamental Python scientific package that allows many high-performance operations on single-dimensional and multidimensional arrays. It also offers many mathematical routines. Of course, it's open-source.

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:

Shell

```
(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.
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. Throughout the rest of the tutorial, you'll learn how to do these steps for several different scenarios.

Step 1: Import packages and classes

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

Python >>>

```
>>> 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:

```
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 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:

```
Python                                                          >>>
>>> 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:

Python >>>

```
>>> 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`:

Python >>>

```
>>> model.fit(x, y)
LinearRegression()
LinearRegression()
```

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 to interpret it.

You can obtain the coefficient of determination, R^2, with `.score()` called on `model`:

Python >>>

```python
>>> 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 R^2.

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

```python
>>> print(f"intercept: {model.intercept_}")
intercept: 5.633333333333329

>>> print(f"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.

Note: In scikit-learn, by convention, a trailing underscore indicates that an attribute is estimated. In this example, `.intercept_` and `.coef_` are estimated values.

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:

```python
>>> 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-

.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()`:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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:

```python
>>> 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()` 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`.

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

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:

```
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                                                                  >>>

>>> 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()`:

```
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(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 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.

In this example, the intercept is approximately 5.52, and this is the value of the predicted response when $x_1 = x_2 = 0$. An increase of x_1 by 1 yields a 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(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:

```python
>>> 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:

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

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:

```
Python                                                                >>>
>>> 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()`:

```
Python                                                                >>>
>>> 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:

```
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                                                           >>>

>>> 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:

```
Python                                                           >>>

>>> 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. 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(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:

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, 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:

```
Python
>>> 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:

```
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(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 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(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:

```
Python                                                                  >>>

>>> # 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:

```
Python                                                                  >>>
>>> 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 $f(x_1, x_2) = b_0 + b_1x_1 + b_2x_2 + b_3x_1^2 + b_4x_1x_2 + b_5x_2^2$.

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 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, check out Split Your Dataset With scikit-learn's train_test_split().

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`:

Python >>>

```python
>>> 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:

Python >>>

```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 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:

```
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                                                                 >>>

>>> 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 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! 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. To find more information about this class, you can visit the official documentation page.

Once your model is created, then 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 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:

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:                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:

```python
>>> 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 R^2.
2. **`.rsquared_adj`** represents adjusted R^2—that is, R^2 corrected according to the number of input features.
3. **`.params`** refers the array with b_0, b_1, and b_2.

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.

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:

Python >>>

```
>>> 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:

Python >>>

```
>>> 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.

Conclusion

You now know what linear regression is and how you can implement it with Python and three open-source packages: NumPy, scikit-learn, and statsmodels. You use NumPy for handling arrays. Linear regression is implemented with the following:

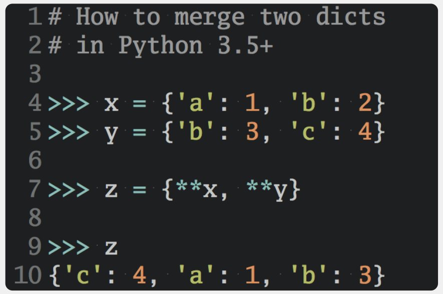
- **scikit-learn** if you don't need detailed results and want to use the approach consistent with other regression techniques
- **statsmodels** if you need the advanced statistical parameters of a model

Both approaches are worth learning how to use and exploring further. The links in this article can be very useful for that.

In this tutorial, you've learned the following steps for performing linear regression in Python:

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**

And with that, you're good to go! If you have questions or comments, please put them in the comment section below.



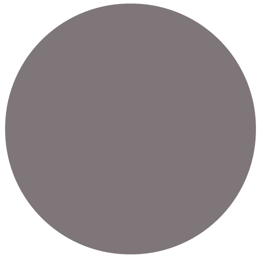
```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

About **Mirko Stojiljković**



Mirko has a Ph.D. in Mechanical Engineering and works as a university professor. He is a Pythonista who applies hybrid optimization and machine learning methods to support decision making in the energy sector.

» More about Mirko

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren Geir Arne Joanna

Kate Kyle

What Do You Think?

Rate this article:

Tweet Share in Share Email

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

Looking for a real-time conversation? Visit the Real Python Community Chat or join the next "Office Hours" Live Q&A Session. Happy Pythoning!

Keep Learning

Related Tutorial Categories: data-science intermediate machine-learning

Recommended Video Course: Starting With Linear Regression in Python