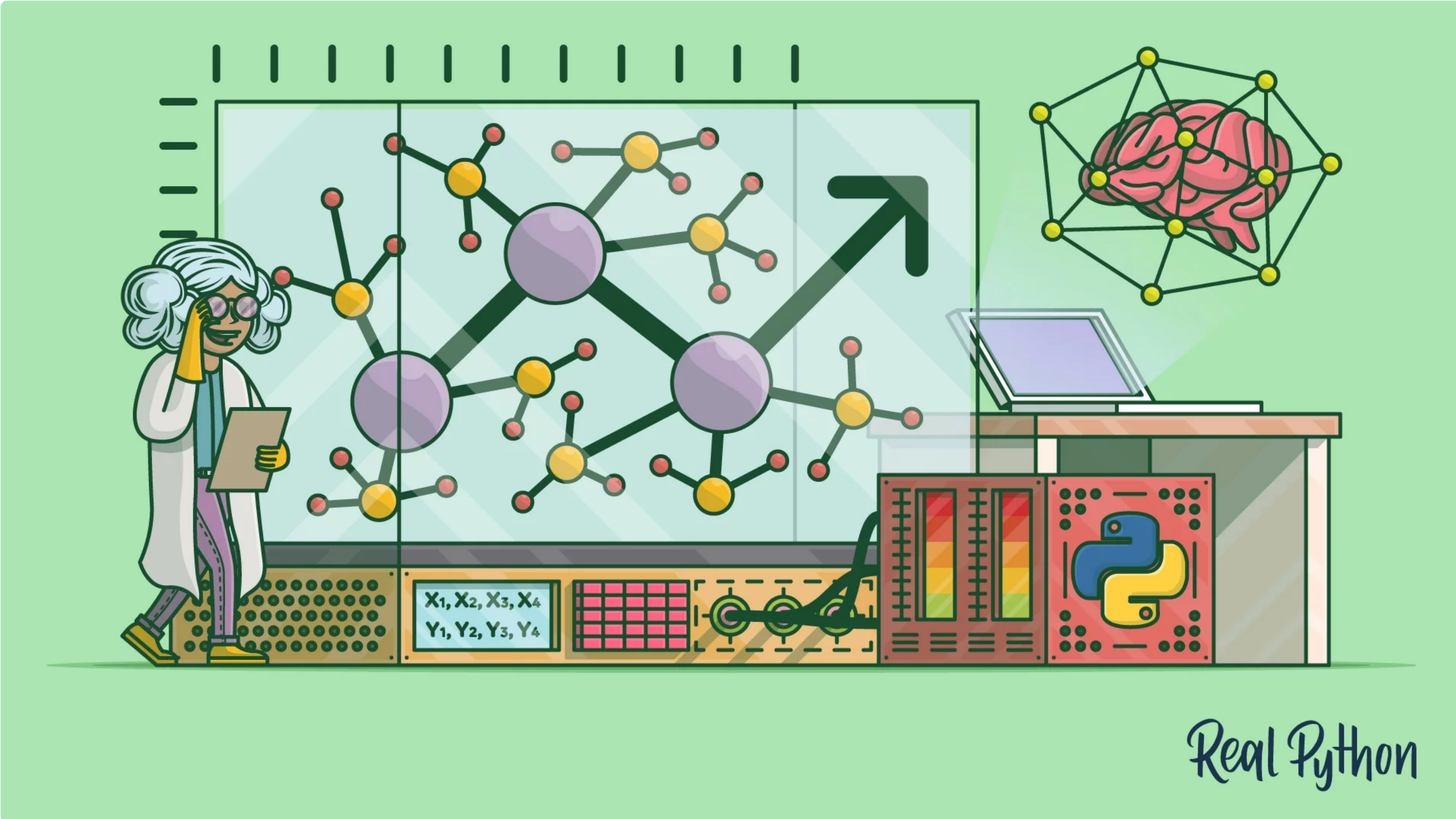


Linear Regression in Python

by Mirko Stojiljkovic Apr 15, 2019 5 Comments data-science intermediate machine-learning

Tweet Share Email

Table of Contents

We’re living in the 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, autonomous vehicles development, 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.

Linear regression is one of the fundamental statistical and machine learning techniques. Whether you want to do statistics, machine learning, or scientific computing, there are good chances that you’ll need it. It’s advisable to learn it first and then proceed towards 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.

Regression

Regression analysis is one of the most in
methods available. Linear regression is

What Is Regression?

Regression searches for relationships an

For example, you can observe several er
on the **features**, such as experience, level of education, role, city they work in, and so on.

This is a regression problem where data related to each employee represent 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 a mathematical dependence of the prices of houses on their areas, numbers of bedrooms, distances to the city center, and so on.

Generally, in regression analysis, you usually 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**, 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, brand, and so on.

It is a common practice to denote the outputs with y and inputs with x. If there are two or more independent variables, 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* the experience or gender impact 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: economy, computer science, social sciences, and so on. 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

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 **predicted weights**, denoted with $b_0, b_1, \ldots, b_r$. They define the **estimated regression function** $f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$. This function should capture the dependencies between the inputs and output sufficiently well.

The **estimated** or **predicted response**, $f($ … corresponding **actual response** y_i. The … Regression is about determining the **bes**… residuals.

To get the best weights, you usually **min**… $\mathrm{SSR} = \Sigma_i(y_i - f(\mathbf{x}_i))^2$. This approach is call…

Regression Performance

The variation of actual responses y_i, $i = 1$… there is also an additional inherent varia… of the output.

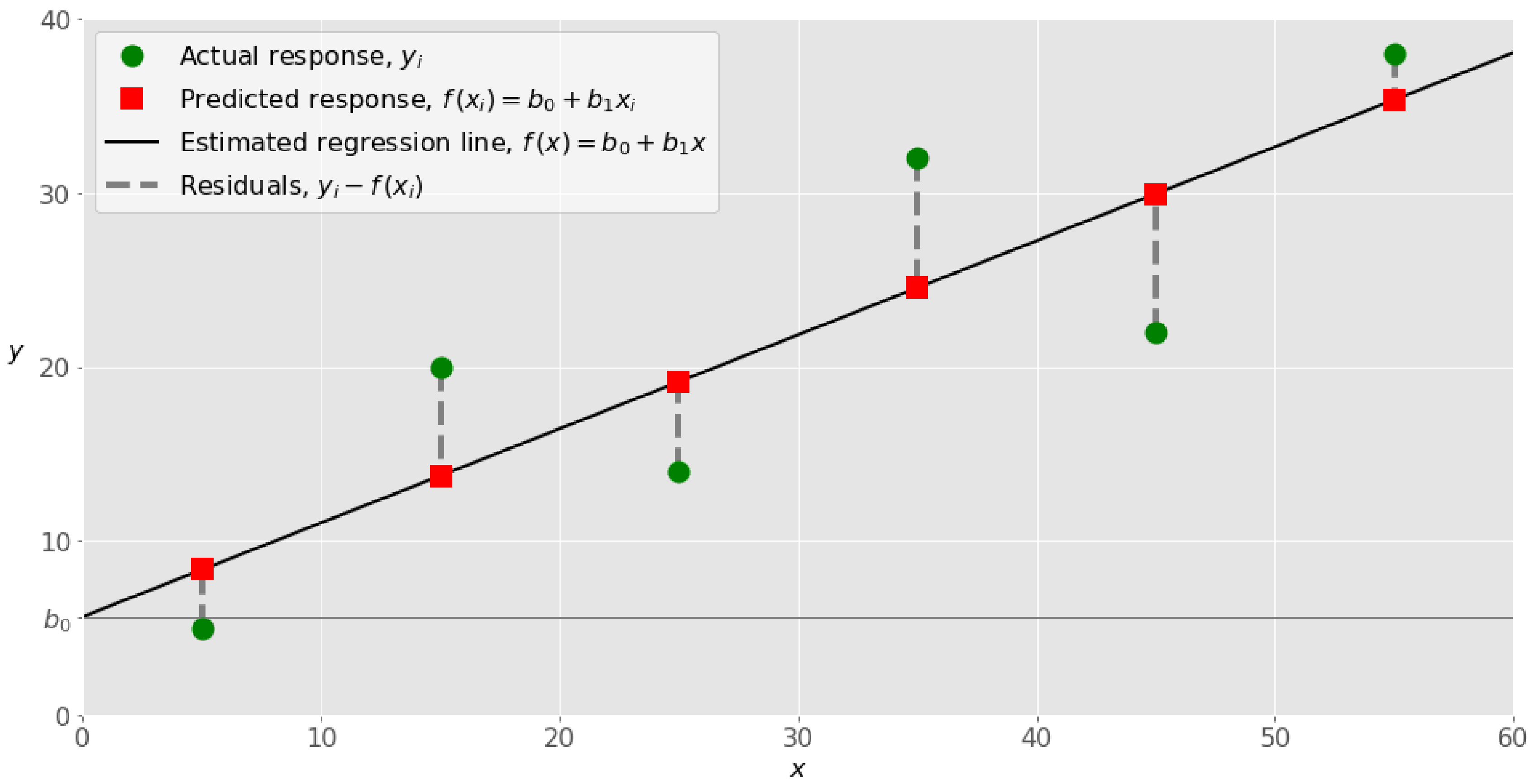
The **coefficient of determination**, denoted as R^2, tells you which amount of variation in y can be explained by the dependence on $\mathbf{x}$ using the particular regression model. Larger R^2 indicates a better fit and means that the model can better explain the variation of the output with different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

The value $R^2 = 1$ corresponds to $\mathrm{SSR} = 0$, that is to 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 with 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 (green circles). These pairs are your observations. For example, the leftmost observation (green circle) has the input $x = 5$ and the actual output (response) $y = 5$. The next one has $x = 15$ and $y = 20$, and so on.

The estimated regression function (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 is 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 (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 $f(5) = 8.33$ (represented with the leftmost red square).

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

Multiple Linear Regressior

Multiple or multivariate linear regressior

If there are just two independent variabl represents a regression plane in a three- weights b_0, b_1, and b_2 such that this plar

The case of more than two independent $f(x_1, \ldots, x_r) = b_0 + b_1x_1 + \cdots + b_rx_r$, 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 non-linear terms such as $b_2x_1^2$, $b_3x_1^3$, or even $b_4x_1x_2$, $b_5x_1^2x_2$, and so on.

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

Now, remember that you want to calculate b_0, b_1, and b_2, which 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 2, 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. What you get as the result of regression are the values of six weights which 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

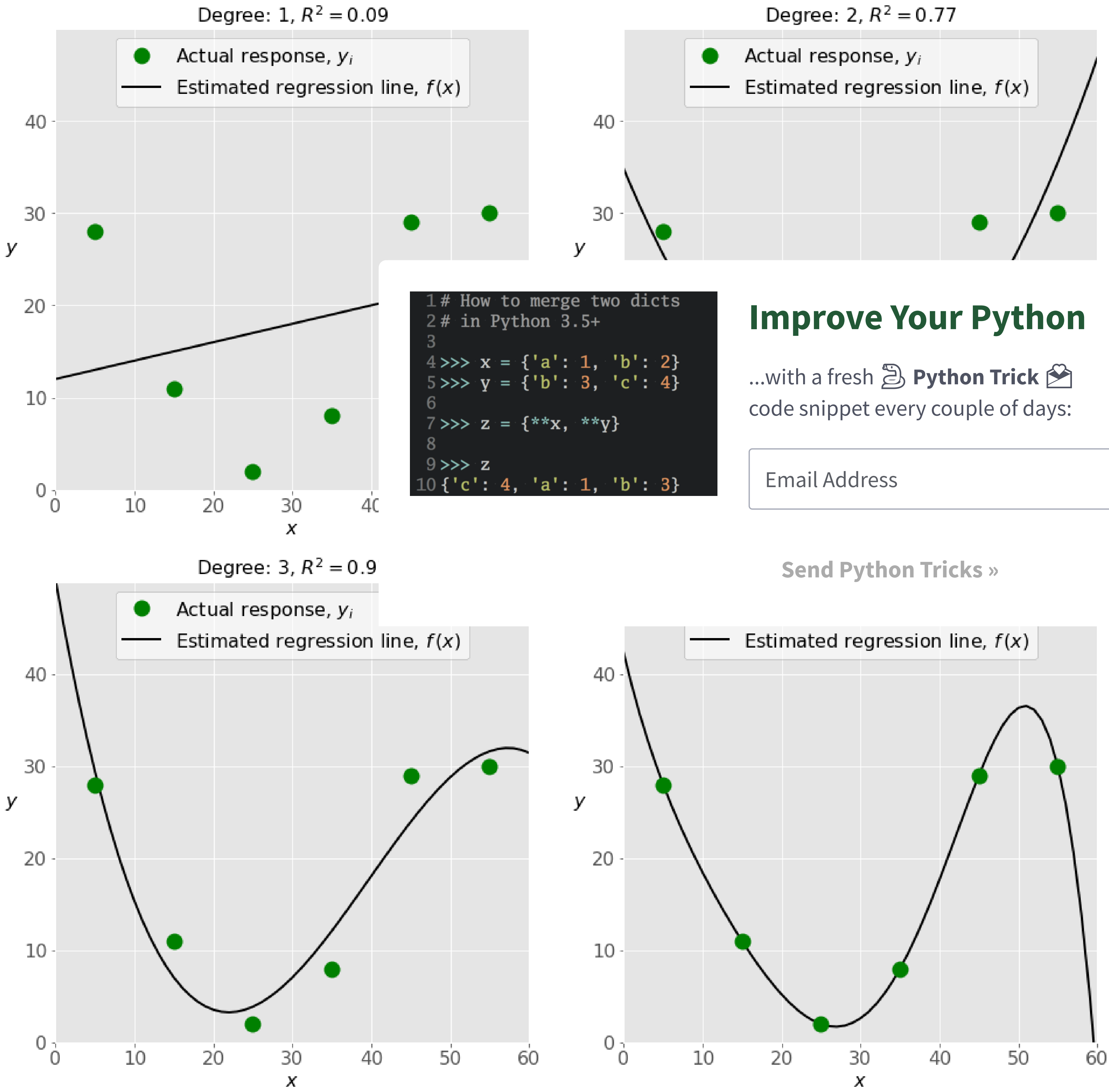
One very important question that might arise when you're implementing polynomial regression is related to **the choice of the optimal degree of the polynomial regression function**.

There is 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 dependencies among data 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 that a straight line can't take into account the fact that the actual response increases as x moves away from 25 towards zero. This is likely an example of underfitting.

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

The bottom left plot presents polynomial regression with the degree equal to 3. 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 60 where the line starts decreasing, although actual data don't show that.

Finally, on the bottom right plot, you can see the perfect fit: six points and the polynomial line of the degree 5 (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 is likely to have poor behavior with unseen data, especially with the inputs larger than 50.

For example, it assumes, without any evidence, that there is a significant drop in responses for $x > 50$ and that y reaches zero for x near 60. 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 article, as well as Khan Academy's linear regression article are good starting points.

Implementing Linear Regression in Python

It's time to start implementing linear regression in Python. Basically, all you should do is apply the proper packages and their functions and classes.

Python Packages for Linea

The package **NumPy** is a fundamental P
single- and multi-dimensional arrays. It

If you're not familiar with NumPy, you ca
Programming With NumPy. In addition,
a pretty good idea on the performance g

The package **scikit-learn** is a widely use
packages. It provides the means for prep
classification, clustering, and more. Like

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

If you want to implement linear regression and need the 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 web site.

Simple Linear Regression With scikit-learn

Let's start with the simplest case, which is simple linear regression.

There are five basic steps when you're implementing linear regression:

1. Import the packages and classes you need.
2. Provide data to work with and eventually do appropriate transformations.
3. Create a regression model and fit it with existing data.
4. Check the results of model fitting to know whether the model is satisfactory.
5. Apply the model for predictions.

These steps are more or less general for most of the regression approaches and implementations.

Step 1: Import packages and classes

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

Python

```python
import numpy as np
from sklearn.linear_model import LinearRegression
```

Now, you have all the functionalities you need to implement linear regression.

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

The class `sklearn.linear_model.LinearRegression` will be used to perform linear and polynomial regression and make predictions accordingly.

Step 2: Provide data

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

Python

```
x = np.array([5, 15, 25, 35, 45, 55]).reshape((-1, 1))
y = np.array([5, 20, 14, 32, 22, 38])
```

Now, you have two arrays: the input `x` and output `y`. You should call `.reshape()` on `x` because this array is required to be **two-dimensional**, or to be more precise, to have **one column and as many rows as necessary**. That's exactly what the argument `(-1, 1)` of `.reshape(`

This is how `x` and `y` look now:

Python

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


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.

Let's create an instance of the class `LinearRegression`, which will represent the regression model:

Python

```
model = LinearRegression()
```

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

- **`fit_intercept`** is a Boolean (`True` by default) that decides whether to calculate the intercept b_0 (`True`) or consider it equal to zero (`False`).
- **`normalize`** is a Boolean (`False` by default) that decides whether to normalize the input variables (`True`) or not (`False`).
- **`copy_X`** is a Boolean (`True` by default) that decides whether to copy (`True`) or overwrite the input variables (`False`).
- **`n_jobs`** is an integer or `None` (default) and represents the number of jobs used in parallel computation. `None` usually means one job and `-1` to use all processors.

This example uses the default values of all parameters.

It's time to start using the model. First, you need to call `.fit()` on `model`:

Python

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

With `.fit()`, you calculate the optimal values of the weights b_0 and b_1, using the existing input and output (`x` and `y`) as the arguments. In other words, `.fit()` **fits the model**. It returns `self`, which is the variable `model` itself. That's why you can replace the last two statements with this one:

Python

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

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

Step 4: Get results

Once you have your model fitted, you can get the results to check whether the model works satisfactorily and interpret it.

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

Python >>>

```
>>> r_sq = model.score(x, y)
>>> print('coefficient of determinat
coefficient of determination: 0.715
```

When you're applying `.score()`, the argu

The attributes of `model` are `.intercept_`,

Python

```
>>> print('intercept:', model.interc
intercept: 5.633333333333329
>>> print('slope:', model.coef_)
slope: [0.54]
```


The code above illustrates how to get b_0 and b_1. You can notice that `.intercept_` is a scalar, while `.coef_` is an array.

The value b_0 = 5.63 (approximately) illustrates that your model predicts the response 5.63 when x is zero. The value b_1 = 0.54 means that the predicted response rises by 0.54 when x is increased by one.

You should notice that you can provide `y` as a two-dimensional array as well. In this case, you'll get a similar result. This is how it might look:

Python >>>

```
>>> new_model = LinearRegression().fit(x, y.reshape((-1, 1)))
>>> print('intercept:', new_model.intercept_)
intercept: [5.63333333]
>>> print('slope:', new_model.coef_)
slope: [[0.54]]
```

As you can see, this example is very similar to the previous one, but in this case, `.intercept_` is a one-dimensional array with the single element b_0, and `.coef_` is a two-dimensional array with the single element b_1.

Step 5: Predict response

Once there is a satisfactory model, you can use it for predictions with either existing or new data.

To obtain the predicted response, use `.predict()`:

Python >>>

```
>>> y_pred = model.predict(x)
>>> print('predicted response:', y_pred, sep='\n')
predicted response:
[ 8.33333333 13.73333333 19.13333333 24.53333333 29.93333333 35.33333333]
```

When applying `.predict()`, you pass the regressor as the argument and get the corresponding predicted response.

This is a nearly identical way to predict the response:

Python >>>

```
>>> y_pred = model.intercept_ + model.coef_ * x
>>> print('predicted response:', y_pred, sep='\n')
predicted response:
[[ 8.33333333]
 [13.73333333]
 [19.13333333]
 [24.53333333]
 [29.93333333]
 [35.33333333]]
```

In this case, you multiply each element (

The output here differs from the previou
dimensional array, while in the previous

If you reduce the number of dimensions
by replacing `x` with `x.reshape(-1)`, `x.fla`

In practice, regression models are often
the outputs based on some other, new i

Python

```
>>> x_new = np.arange(5).reshape((-1, 1))
>>> print(x_new)
[[0]
 [1]
 [2]
 [3]
 [4]]
>>> y_new = model.predict(x_new)
>>> print(y_new)
[5.63333333 6.17333333 6.71333333 7.25333333 7.79333333]
```

Here `.predict()` is applied to the new regressor `x_new` and yields the response `y_new`. This example conveniently uses `arange()` from `numpy` to generate an array with the elements from 0 (inclusive) to 5 (exclusive), that is `0`, `1`, `2`, `3`, and `4`.

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.

Steps 1 and 2: Import packages and classes, and provide data

First, you import `numpy` and `sklearn.linear_model.LinearRegression` and provide known inputs and output:

Python

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

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

That's a simple way to define the input `x` and output `y`. You can print `x` and `y` to see how they look now:

Python >>>

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

In multiple linear regression, x is a two-c
dimensional array. This is a simple exam

Step 3: Create a model and fit it

The next step is to create the regression

Python

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


The result of this statement is the variable `model` referring to the object of type `LinearRegression`. It represents the regression model fitted with existing data.

Step 4: Get results

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

Python >>>

```
>>> r_sq = model.score(x, y)
>>> print('coefficient of determination:', r_sq)
coefficient of determination: 0.8615939258756776
>>> print('intercept:', model.intercept_)
intercept: 5.52257927519819
>>> print('slope:', model.coef_)
slope: [0.44706965 0.25502548]
```

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

In this example, the intercept is approximately 5.52, and this is the value of the predicted response when $x_1 = x_2 = 0$. The increase of x_1 by 1 yields the rise of the predicted response by 0.45. Similarly, when x_2 grows by 1, the response rises by 0.26.

Step 5: Predict response

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

Python >>>

```
>>> y_pred = model.predict(x)
>>> print('predicted response:', y_pred, sep='\n')
predicted response:
[ 5.77760476  8.012953   12.73867497 17.9744479  23.97529728 29.4660957
 38.78227633 41.27265006]
```

The predicted response is obtained with `.predict()`, which is very similar to the following:

Python >>>

```
>>> y_pred = model.intercept_ + np.sum(model.coef_ * x, axis=1)
>>> print('predicted response:', y_pred, sep='\n')
predicted response:
[ 5.77760476  8.012953   12.73867497 17.9744479  23.97529728 29.4660957
 38.78227633 41.27265006]
```

You can predict the output values by multiplying each column of the input with the appropriate weight, summing the results and adding the intercept to the sum.

You can apply this model to new data as well:

Python

```
>>> x_new = np.arange(10).reshape((-
>>> print(x_new)
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
>>> y_new = model.predict(x_new)
>>> print(y_new)
[ 5.77760476  7.18179502  8.5859852
```


That's the prediction using a linear regression model.

Polynomial Regression With scikit-learn

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

Step 1: Import packages and classes

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

Python

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

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

Step 2a: Provide data

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

Python

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

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

Step 2b: Transform input data

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

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

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

Python

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

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

You can provide several optional parameters to `PolynomialFeatures`:

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

This example uses the default values of a the function, and it can be beneficial to p

Before applying `transformer`, you need t

Python

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

Once `transformer` is fitted, it's ready to create a new, modified input. 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)
```

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

Python >>>

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

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

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

Step 3: Create a model and fit it

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

Python

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

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

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

Step 4: Get results

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

Python

```
>>> r_sq = model.score(x_, y)
>>> print('coefficient of determination:', r_sq)
coefficient of determination: 0.8908516262498564
>>> print('intercept:', model.intercept_)
intercept: 21.372321428571425
>>> print('coefficients:', model.coe
coefficients: [-1.32357143  0.028392
```

Again, `.score()` returns R^2. Its first argur associated to `.intercept_` and `.coef_`: . and b_2 respectively.

You can obtain a very similar result with

Python

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

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

Python

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

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

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

Python

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

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

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

Python

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

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

Step 5: Predict response

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

Python

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

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

You can apply the identical procedure if
one column, but everything else is the s

Python

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

# Step 2a: Provide data
x = [[0, 1], [5, 1], [15, 2], [25, 5], [35, 11], [45, 15], [55, 34], [60, 35]]
y = [4, 5, 20, 14, 32, 22, 38, 43]
x, y = np.array(x), np.array(y)

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

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

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

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

This regression example yields the following results and predictions:

Python

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

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

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

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

Advanced Linear Regression With `statsmodels`

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

The procedure is similar to that of scikit-learn.

Step 1: Import packages

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

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


Now you have the packages you need.

Step 2: Provide data and transform in

You can provide the inputs and outputs

Python
```
x = [[0, 1], [5, 1], [15, 2], [25, 5], [35, 11], [45, 15], [55, 34], [60, 35]]
y = [4, 5, 20, 14, 32, 22, 38, 43]
x, y = np.array(x), np.array(y)
```

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

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

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

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

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

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

Step 3: Create a model and fit it

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

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

You should be careful here! Please, notice that the first argu

several more optional parameters.

To find more information about this class, please visit [the official documentation page](#).

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

Python

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

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

Step 4: Get results

The variable `results` refers to the object

Explaining them is far beyond the scope

You can call `.summary()` to get the table w

Python

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

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly 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 the warning related to `kurtosistest`. This is due to the small number of observations provided.

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

Python >>>

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

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

1. `.rsquared` holds R^2.

2. `.rsquared_adj` represents adjusted R^2 (R^2 corrected according to the number of input features).
3. `.params` refers the array with b_0, b_1, and b_2 respectively.

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 ar

Python

```
>>> print('predicted response:', res
predicted response:
[ 5.77760476  8.012953   12.73867497
 38.78227633 41.27265006]
>>> print('predicted response:', res
predicted response:
[ 5.77760476  8.012953   12.73867497
 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)))
>>> print(x_new)
[[1. 0. 1.]
 [1. 2. 3.]
 [1. 4. 5.]
 [1. 6. 7.]
 [1. 8. 9.]]
>>> y_new = results.predict(x_new)
>>> print(y_new)
[ 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 non-linear 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 the 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.

When performing linear regression in Python, you can follow these steps:

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 t
5. Apply the model for predictions

If you have questions or comments, plea


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

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

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

Email Address

Send Me Python Tricks »

About **Mirko Stojiljkovic**

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 decisio 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

Joanna

Kyle

What Do You Think?

Real Python Comment Policy: Th
from or helping out other readers–
Complaints and insults generally v

Keep Learning

Related Tutorial Categories: data-science intermediate machine-learning

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end intermediate machine-learning python testing tools web-dev web-scraping

```
# 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}
```


Table of Cor