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 statsmo
```

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

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])
```

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()
```

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.