Getting Started

Five Regression Python Modules That Every Data Scientist Must Know

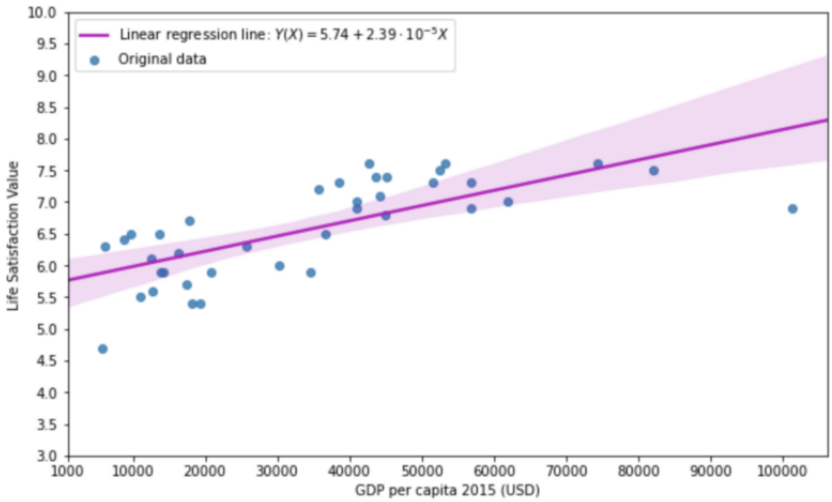

Fig. 1. Plot of life satisfaction value versus GDP per capita by using the **seaborn** python library (figure created by the author for educational purposes) as in section 5. The colored region represents the 95% confidence region of the linear regression line.

Introduction

Regression is a very important concept in statistical modelling, data science, and machine learning that helps establish a possible relationship between an independent variable (or predictor), $\boldsymbol{x}$, with a dependent variable (or simply output) $y(\boldsymbol{x})$ by using specific mathematical minimisation criteria. There are several types of regression that are used in different situations and one of the most common is linear regression. Other types of regression include logistic regression, non-linear regression, etc.

In Python, there are several libraries and corresponding modules that can be used to perform regression depending on a specific problem that one encounters and its complexity. In this article, I will summarise the five most important modules and libraries in Python that one can use to perform regression and also will discuss some of their limitations. Here I assume that the reader knows Python and some of its most important libraries.

1. polyfit of NumPy

NumPy that stands for *Numerical Python* is probably the most important and efficient Python library for numerical calculations involving arrays. In addition to several operations for numerical calculations, NumPy has also a module that can perform simple linear regression and polynomial regression. To make things more clear it is

better to give a specific example involving NumPy arrays that represent realistic data as below:

```
import numpy as np
x = np.array([ 56755.72171242,  44178.04737774,  40991.80813814,   8814.00098681, 43585.51198178,  13574.17183072,  61
y = np.array([7.3, 7.1, 6.9, 6.4, 7.4, 6.5, 6.3, 6.7, 7.6, 5.7, 7.6, 6.5, 7.0, 5.4, 5.6, 7.5, 7.0, 7.2, 6.0, 5.9, 5.9, 5.9, 6.9, 6.5, 7.4, 7.3, 7
```

The NumPy array x represents the GDP per capita in USD for a given country and the array y represents the life satisfaction value of people in a given country. The life satisfaction value is in the range [0, 10] where a value of 10 corresponds to a maximum satisfaction while a value of 0 is the total absence of satisfaction. The details of what is the relation between life satisfaction and GDP per capita for several countries can be found on my GitHub page.

As mentioned above, the NumPy library has an option that gives the user the possibility to perform a linear regression (simple and polynomial) by using the least square method as minimization criteria. The module that does this regression is polyfit: **np.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False).** The x array is of shape (M,) while the y array is of shape (M, K) where M and K are positive natural numbers. In addition, **np.polyfit()** gives the possibility to specify the degree of polynomial regression with the "**deg = n**" and also can calculate the *covariance matrix* that gives important information about the coefficients of the polynomial regression. **polyfit**, fits the data by using the least square method and internally memorises the coefficients of the linear regression found during the fitting procedure. To plot the linear regression function one needs to convert the already found polynomial coefficients into a polynomial function through the function **np.poly1d().**

As an example, now I use the **np.polyfit()** function to perform a simple linear regression (*n = 1*) on the x and y arrays above and plot the result. I use the following Python code:

```
[In]: import matplotlib.pyplot as plt
[In]: p = np.poly1d(np.polyfit(x, y, 1))
[In]: x_line = np.linspace(np.amin(x), np.amax(x), 200)
[In]: plt.scatter(x, y)
[In]: plt.plot(x_line, p(x_line))
[In]: plt.show()
```

You can run the above Python code in your computer to show the plot of the simple linear regression, however, here I do not show the plot for sake of clarity. Moreover, **polyfit** gives the user the possibility to know the coefficients of the linear regression. Indeed, if you display the variable p in the above code you will get the following linear regression line with the equation:

```
[In]: print(p)
```

[Out]: 2.4e-05 x + 5.742

So, the linear regression with **np.polyfit()** gave as a result a linear regression line ($y(x) = a + bx$) with intercept, a=5.741 (precise value), and slope, b =2.39e-05 (precise value). The print(p) command gives an approximate value display.

The **polyfit** module is very useful for fitting simple linear regression and polynomial regression of degree n. However, it does not give the user the possibility to use linear regression with multiple predictor variables, namely multivariate regression. So, it is not possible to use **np.polyfit()** for mixed interaction terms but only for self-interaction terms. In addition, it does not give the user the possibility to *directly* calculate: the coefficient of determination R^2 to assess the goodness of the fit, the Pearson correlation coefficient r, the p-value of hypothesis testing, and sample errors associated with the regression coefficients.

2. linregress of SciPy

SciPy is a Python library that stands for *Scientific Python*. It is the most important library for scientific computing that is used in academia and the scientific industry. This library contains several modules that are used for specific purposes. Among these modules, the **scipy.stats()** module, is the most important one regarding statistical modelling in general. The **scipy.stats()** module has a submodule completely dedicated to linear regression which goes under the syntax: **scipy.stats.linregress()** and uses the least square method as a minimisation criteria.

Now to see **linregress** in action, I use again the arrays x and y as above and use the following Python code:

```
[In]: import scipy as sp
[In]: regr_results = sp.stats.linregress(x, y)
[In]: print(regr_results)
```

[Out]: LinregressResult(slope=2.3996299825729513e-05, intercept=5.741754353755326, rvalue=0.720287195322656, pvalu

As you can see from the above Python code, the **linregress** module gives as an output the results of the linear regression, where the intercept value is, a = 5.741 and, the slope value is b = 2.39e-05. These values of a and b are the same as those found by using the **polyfit** module of NumPy as in the previous section. In addition, **linregress** evaluates the Pearson correlation coefficient r (with value of rvalue = 0.72), the p-value (pvalue = 3.42e-06), the standard deviation of the slope b (stderr = 3.85e-06), and the standard error of the intercept term a (intercept_stderr = 0.15). Explicit calculations and Python codes can be found in my GitHub page.

The **linregress** module gives additional results of the linear regression

to the **polyfit** module as shown above. The only disadvantage of **linregress** is that does not support multivariate regression. It only supports simple linear regression. In addition, it does not give to the user the option to *directly* predict new values for features not used in the least square method like the scikit-learn library as in section 5 below.

3. OLS and ols of statsmodels

The **statsmodels** library/module is an extension of the **scipy.stats** module that is mainly used for fitting a model to a given dataset. This module is probably the most complete one regarding regression in general and also linear regression in particular. This module is quite flexible and it gives the user several options to perform specific statistical calculations.

As I did in sections 2 and 3, I use the **statsmodel** to perform a simple linear regression by using the x, and y arrays as above and using the least square method as minimisation criteria with the **OLS** module. I use the following Python code:

```
[In]: import statsmodels.api as sm
[In]: x = sm.add_constant(x) # Adds an intercept term to the simple linear regression formula
[In]: lin_model = sm.OLS(y, x)
[In]: regr_results = lin_model.fit()
[In]: print(regr_results.results)

[Out]: [5.74175435e+00 2.39962998e-05]
```

After printing the results with the above code, I get the following values for intercept a = 5.741 and slope b = 2.39e-05 of the simple linear regression on x and y arrays. The **OLS** module implicitly uses the least square minimisation method for calculating the regression coefficients. One can note that the values of a and b coincide with those previously found in sections 1 and 2 with other methods.

A more detailed description of the regression results can be obtained with the python command **print(regr_results.summary())** where the results table is shown in Fig. 2. As you can see, the summary table gives a detailed information of the linear regression results that include: the coefficient of determination R^2, the value of the intercept a and its standard deviation, the value of the slope b and its standard deviation, the value of the t score, the p-value, the confidence interval, etc.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.519
Model:                            OLS   Adj. R-squared:                  0.505
Method:                 Least Squares   F-statistic:                     38.82
Date:                Mon, 23 Aug 2021   Prob (F-statistic):           3.43e-07
Time:                        23:29:37   Log-Likelihood:                -28.824
No. Observations:                  38   AIC:                             61.65
Df Residuals:                      36   BIC:                             64.92
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.7418      0.159     36.218      0.000       5.420       6.063
x1           2.4e-05   3.85e-06      6.230      0.000    1.62e-05    3.18e-05
==============================================================================
Omnibus:                        2.630   Durbin-Watson:                   1.765
Prob(Omnibus):                  0.268   Jarque-Bera (JB):                2.372
Skew:                          -0.591   Prob(JB):                        0.305
Kurtosis:                       2.680   Cond. No.                     7.58e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 7.58e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Fig. 2. Results table of the simple linear regression by using the **OLS** module of the **statsmodel** library.

The **OLS** module and its equivalent module, **ols** (I do not explicitly discuss about **ols** module in this article) have an advantage to the **linregress** module since they can perform multivariate linear regression. On the other hand, the disadvantage of the module **ols,** is that it does not have the option to *directly* predict new values *y* for new values of predictors ***x_i*** (at least not known to me). Also, another disadvantage of the **OLS** module is that one has to add explicitly a constant term for the linear regression with the command **sm.add_constant()**. The **linear_model.OLS** module, on the other hand, gives the user the possibility to predict new values given a design matrix similar to the **LinearRegression** module of scikit-learn.

4. LinearRegression of scikit-learn

scikit-learn is one of the best Python libraries for statistical/machine learning and it is adapted for fitting and making predictions. It gives the user different options for numerical calculations and statistical modelling. Its most important sub-module for linear regression is **LinearRegression.** It uses the least square method as minimisation criteria to find the parameters of the linear regression.

As I did in the previous sections, I use the arrays x and y as above for simple linear regression. I use the following Python code:

```
[In]: from sklearn import linear_model
[In]: linmodel = linear_model.LinearRegression(fit_intercept=True)
[In]: linmodel.fit(x.reshape(-1, 1), y)
[Out]: LinearRegression()
```

The above Python code uses linear regression to fit the data contained in the x and y arrays. If now one needs to get some of the parameters from the fit, it is necessary to write an additional code. For example, I want to see the values of intercept *a* and slope *b* of the fitting procedure. To do this, I run the following Python code:

```
[In]: (slope, intercept) = (model_1.coef_[0], model_1.intercept_)
[In]: print(slope, intercept)
[Out]: 2.3996299825729496e-05 5.741754353755327
```

As you can see, the **LinearRegression** module gives the same values of intercept *a* and slope *b* as previously found by using other methods. In addition, it is also possible to calculate the coefficient of determination R^2 with the Python command*:* **print(linmodel.score(x.reshape(-1, 1), y))** which gives a value of R^2 = 0.518 that is the same as that given by using **OLS** module results of Fig. 2.

The advantage of the **LinearRegression** module is that it gives the user the possibility to *directly* predict new values for new data with the **linmodel.predict()** command. This function makes the **LinearRegression** module very appealing for statistical/machine learning. As the **OLS** module, the **LinearRegression** module can also perform multivariate linear regression if needed. The disadvantage of **LinearRegression** module is that it does not have a summary table of the regression results as the **OLS** module and it forces the user to explicitly write new commands to get important statistical information. Also, it can be quite cumbersome to use the **LinearRegression** module for polynomial regression since one needs to calculate the design matrix $\boldsymbol{X}$ before getting the regression results. One can see this explicitly in my previous article.

5. regplot of seaborn

The seaborn Python library is a very important library for visualisation of statistical results. Technically, it is not a library that can be used to calculate the regression parameters as I showed in the previous sections, but it can be used to graphically visualise the regression lines and confidence regions in a plot. For example, if I want to plot the simple linear regression line obtained in the previous sections, I need to run the following Python code:

```
[In]: import seaborn as sns
[In]: import matplotlib.pyplot as plt

[In]: fig, ax = plt.subplots(figsize=(10, 6))
sns.regplot(x = x, y = y, ci=95, order=1,line_kws={'label': 'Linear regression line: $Y(X)=5.74+2.39\cdot 10^{-5} X$', 'color':
ax.set_xlabel("GDP per capita 2015 (USD)")
ax.set_ylabel("Life Satisfaction Value")
ax.set_xticks([1000, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000])
ax.set_yticks(np.arange(3.0, 10.5, 0.5))
ax.legend(loc="upper left")
```

The result of the above code gives exactly Fig. 1 as shown at the top of this article. The **regplot** module of seaborn internally calculates the values of the linear regression parameters and it plots the linear regression line with the 95% confidence zone (I set the parameter “ci=95” in the above code) of the linear regression parameters. Also, the

regplot module can perform graphical visulalization of multivariate linear regression and logistic regression since this module is strongly based on the **statsmodel** library. The only disadvantage of seaborn in general is that it does not give to the user the possibility to *directly* see the regression results with a specific Python command.

Conclusion

In this article, I summarised the most import python libraries and their modules for regression and I gave specific examples for linear regression. The advantage of a module over another one depends on a specific problem that the user faces. For simple linear regression and polynomial regression, the **polyfit** and **linregress** modules are the easiest to use and very handy. On the other hand for detailed statistical results of linear regression, the **OLS** and **ols** modules are probably the best since they are not difficult to use and give plenty of regression results in a summary table. Also, the **OLS** sub-module of the **linear_model**, gives the user the possibility to make prediction as well with the help of the **linear_model.OLS** module**.** For statistical/machine learning, the **LinearRegression** module of the scikit-learn Python library is one of the best since it can be used to make predictions, a functionality that the majority of other mentioned modules above do not have. If ones desires to plot the results of a statistical procedure directly without information of the fitting parameter values, then the **regplot** module of seaborn is one of the best.

In my summary of the best Python modules for regression, I did not include the Pandas library even though it is possible to calculate some parameters of linear regression explicitly such as the Pearson coefficient *r* etc., see my GitHub page for details. Also, the **lsqt** module of NumPy gives the possibility to do some regression but both Pandas library and **lsqt** module are at an inferior level compared to the modules mentioned in the above sections.

If you liked my article, please share it with your friends that might be interested in this topic and cite/refer to my article in your research studies. Do not forget to subscribe for other related topics that will post in the future.

By Damian Ejlli on August 24, 2021.

Canonical link

Exported from Medium on October 8, 2021.