ORACLE + DATASCIENCE.COM

7 Effective Methods for Fitting a Linear Model in Python

Author: Tirthajyoti Sarkar / Posted on April 2, 2018

This article is republished with permission from the author from Medium's Towards Data Science blog. View the original here.

For a myriad of data scientists, linear regression is the starting point of many statistical modeling and predictive analysis projects. The importance of fitting, both accurately and quickly, a linear model to a large data set cannot be overstated.

With Python fast emerging as the de-facto programming language of choice, it is critical for a data scientist to be aware of all the various methods he or she can use to quickly fit a linear model to a fairly large data set and assess the relative importance of each feature in the outcome of the process.

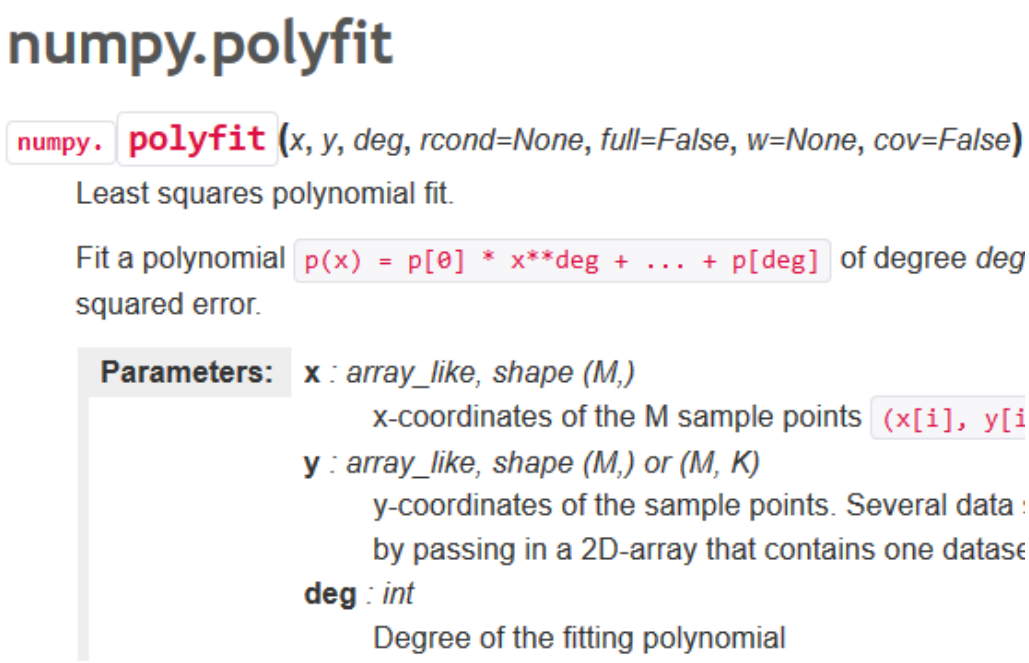
So, in the case of multiple available options, how do you choose the most effective method?

Due to the popularity of scikit-learn, a free machine learning library that features functions for regression, classification, clustering, model selection, and dimensionality reduction, one common approach is calling the Linear Model class from that library and fitting the data. While this can offer additional advantages of applying other pipeline features of machine learning (e.g. data normalization, model coefficient regularization, feeding the linear model to another downstream model), this is often not the fastest or cleanest method when a data analyst needs just a quick and easy way to determine the regression coefficients (and some basic associated statistics).

Below are 7 other methods that are faster and cleaner, though they don't all offer the same amount of information or modeling flexibility. I will discuss each method briefly.

Note: The entire boiler plate code for various linear regression methods is available here on my GitHub repository. Most of them are based on the SciPy package.

1. Method: Scipy.polyfit( ) or numpy.polyfit( )



numpy.polyfit

`numpy.` **`polyfit`** (*x, y, deg, rcond=None, full=False, w=None, cov=False*)

Least squares polynomial fit.

Fit a polynomial `p(x) = p[0] * x**deg + ... + p[deg]` of degree *deg* squared error.

Parameters:

x : *array_like, shape (M,)*
x-coordinates of the M sample points `(x[i], y[i`

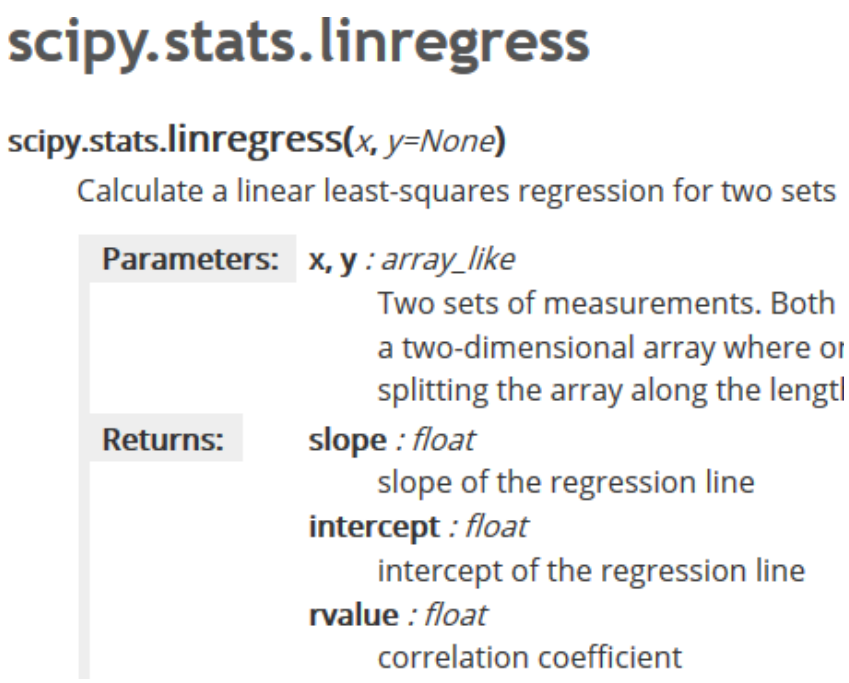
y : *array_like, shape (M,) or (M, K)*
y-coordinates of the sample points. Several data
by passing in a 2D-array that contains one datase

deg : *int*
Degree of the fitting polynomial

This is a pretty general least squares polynomial fit function which accepts the data set and a polynomial function of any degree (specified by the user), and returns an array of coefficients that minimizes the squared error. A detailed description of the function is given here. For simple linear regression, one can choose degree 1. If you want to fit a

model of higher degree, you can construct polynomial features out of the linear feature data and fit to the model too.

2. Method: Stats.linregress( )



scipy.stats.linregress

scipy.stats.**linregress**(*x*, *y=None*)

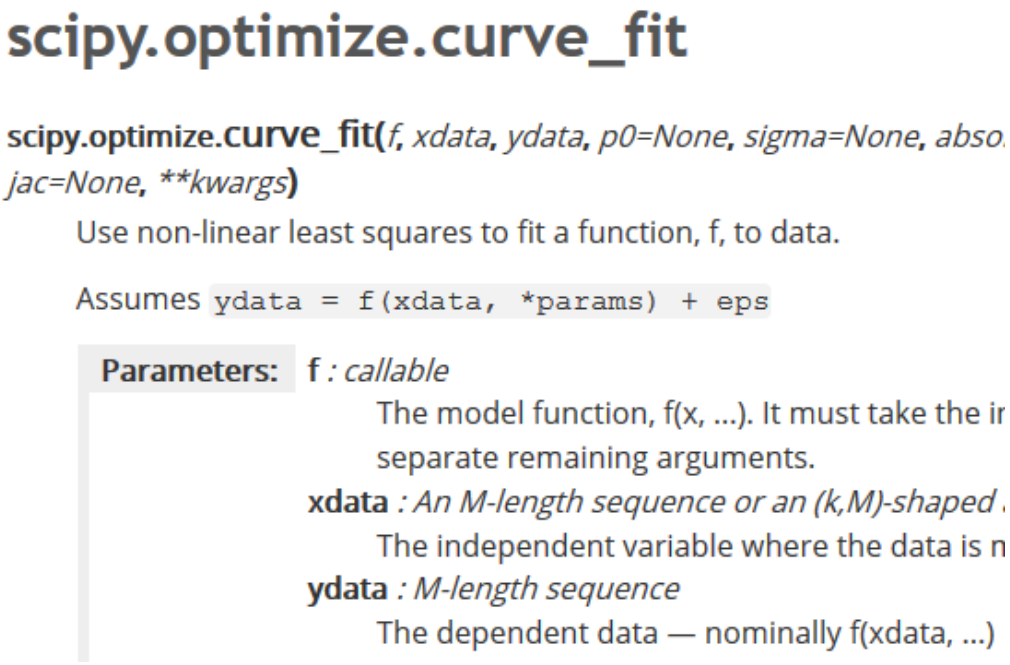
Calculate a linear least-squares regression for two sets

Parameters: **x, y** *: array_like*
Two sets of measurements. Both
a two-dimensional array where o
splitting the array along the lengt

Returns: **slope** *: float*
slope of the regression line
intercept *: float*
intercept of the regression line
rvalue *: float*
correlation coefficient

This is a highly specialized linear regression function available within the stats module of Scipy. It is fairly restricted in its flexibility as it is optimized to calculate a linear least-squares regression for two sets of measurements only. Thus, you cannot fit a generalized linear model or multi-variate regression using this. However, because of its specialized nature, it is one of the fastest method when it comes to simple linear regression. Apart from the fitted coefficient and intercept term, it also returns basic statistics such as R^2 coefficient and standard error.

3. Method: Optimize.curve_fit( )



scipy.optimize.curve_fit

scipy.optimize.curve_fit(*f*, *xdata*, *ydata*, *p0=None*, *sigma=None*, *abso*
jac=None, ***kwargs*)

Use non-linear least squares to fit a function, f, to data.

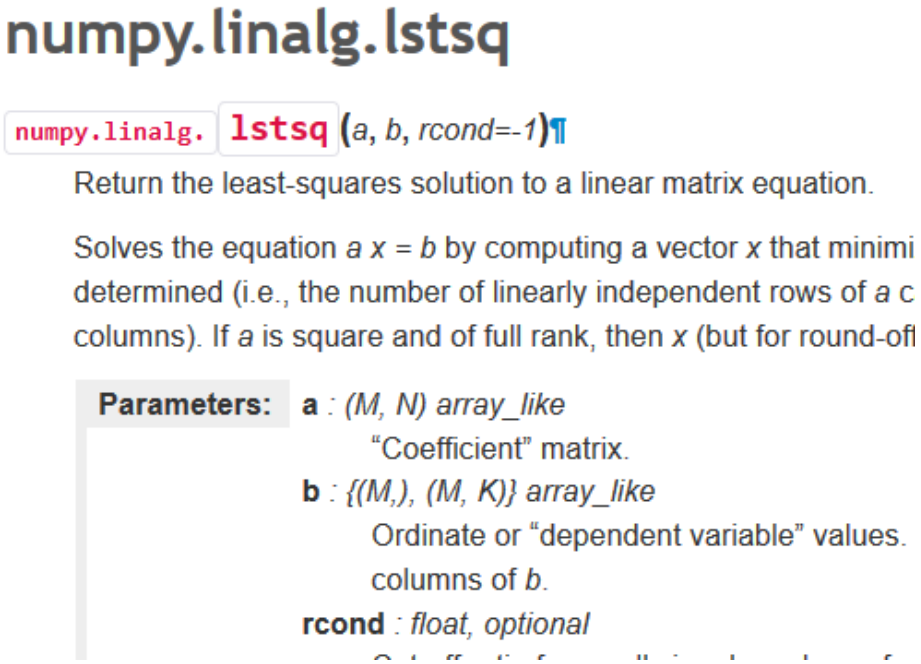
Assumes `ydata = f(xdata, *params) + eps`

Parameters: **f** *: callable*
The model function, f(x, ...). It must take the ir
separate remaining arguments.
xdata *: An M-length sequence or an (k,M)-shaped*
The independent variable where the data is n
ydata *: M-length sequence*
The dependent data — nominally f(xdata, ...)

This is along the same lines as the Polyfit method, but more general in nature. This powerful function from scipy.optimize module can fit any user-defined function to a data set by doing least-square minimization.

For simple linear regression, one can just write a linear mx+c function and call this estimator. It goes without saying that this works for a multivariate regression as well. This method returns an array of function parameters for which the least-square measure and the associated covariance matrix is minimized .

4. Method: numpy.linalg.lstsq



numpy.linalg.lstsq

numpy.linalg. lstsq (a, b, rcond=-1)¶

Return the least-squares solution to a linear matrix equation.

Solves the equation a x = b by computing a vector x that minimi
determined (i.e., the number of linearly independent rows of a c
columns). If a is square and of full rank, then x (but for round-ofl

Parameters: a : (M, N) array_like
"Coefficient" matrix.
b : {(M,), (M, K)} array_like
Ordinate or "dependent variable" values.
columns of b.
rcond : float, optional

This is the fundamental method of calculating least-square solution to a linear system of equation by matrix factorization. It comes from the handy linear algebra module of numpy package. Under the hood, it solves the equation $a\ x = b$ by computing a vector x that minimizes the Euclidean 2-norm $|| b - a\ x ||^2$.

The equation may be under-, well-, or over- determined (i.e., the number of linearly independent rows of *a* can be less than, equal to, or greater than its number of linearly independent columns). If *a* is square and of full rank, then *x* (but for round-off error) is the "exact" solution of the equation.

You can do either a simple or multivariate regression with this and get back the calculated coefficients and residuals. One little trick is that before calling this function you have to append a column of 1's to the x data to calculate the intercept term. Turns out, this is one of the faster methods to try for linear regression problems.

5. Method: Statsmodels.OLS ( )

Statsmodels is a great little Python package that provides classes and functions for estimating different statistical models, as well as conducting statistical tests and statistical

data exploration. An extensive list of result statistics are available for each estimator. The results are tested against existing statistical packages to ensure correctness.

For linear regression, one can use the OLS or Ordinary-Least-Square function from this package and obtain the full-blown statistical information on the estimation process.

One little trick to remember is that you have to add a constant manually to the x data for calculating the intercept, otherwise by default it will report the coefficient only. Below is the snapshot of the full results summary of the OLS model. As you can see, it is as rich as any functional statistical language like R or Julia.

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       1.000
Model:                            OLS   Adj. R-squared:                  1.000
Method:                 Least Squares   F-statistic:                 9.772e+33
Date:                Fri, 08 Dec 2017   Prob (F-statistic):               0.00
Time:                        22:38:16   Log-Likelihood:             2.6723e+08
No. Observations:            10000000   AIC:                        -5.345e+08
Df Residuals:                9999998   BIC:                        -5.345e+08
Df Model:                          1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -6.5000    1.9e-16  -3.42e+16      0.000      -6.500      -6.500
x1             3.2500   3.29e-17   9.89e+16      0.000       3.250       3.250
==============================================================================
Omnibus:                  8794275.792   Durbin-Watson:                   0.000
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           598902.461
Skew:                          -0.000   Prob(JB):                         0.00
Kurtosis:                       1.801   Cond. No.                         5.77
==============================================================================
```

6. Method: Analytic solution using matrix inverse method

For well-conditioned linear regression problems (at least where # of data points > # of features), a simple closed-form matrix solution exists for calculating the coefficients which guarantees least-square minimization. It is given by,

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}.$$

Detailed derivation and discussion about this solution is discussed here.

One has two choices here:

(a) Using simple multiplicative matrix inverse, or

(b) Computing the Moore-Penrose generalized pseudoinverse matrix of x-data followed by taking a dot product with the y-data. Because this second process involves singular-value decomposition (SVD), it is slower but works well for data sets that are not well-conditioned .

7. Method: sklearn.linear_model.LinearRegression( )

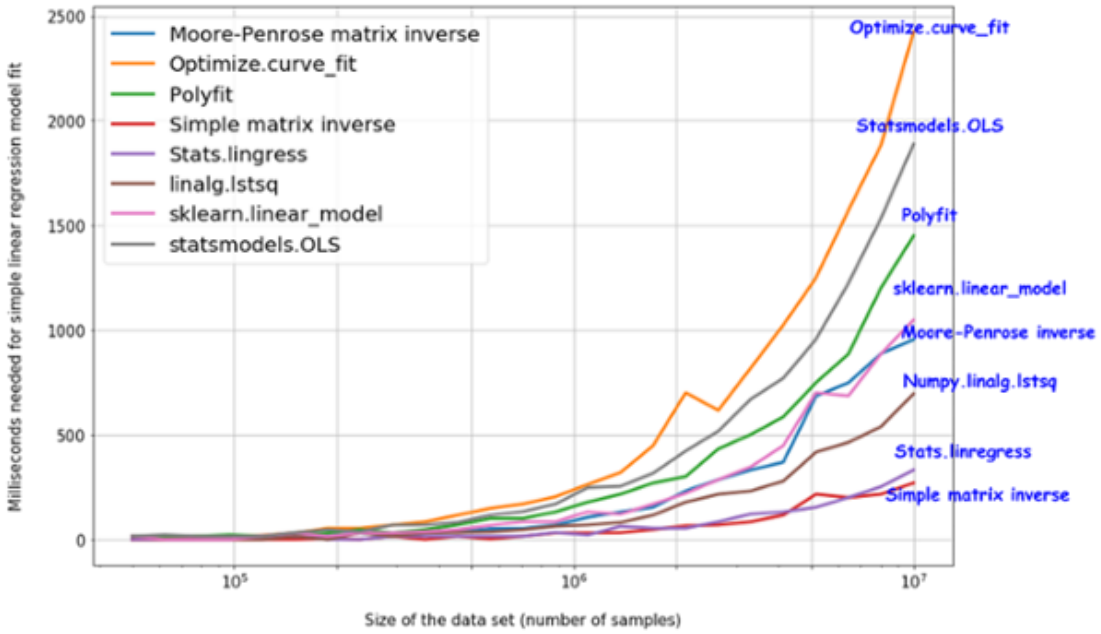
This is the quintessential method used by the majority of machine learning engineers and data scientists. Of course, for real world problems, it is usually replaced by cross-validated and regularized algorithms, such as Lasso regression or Ridge regression. However, the essential core of those advanced functions lies in this model.

Measuring speed and time complexity of these methods

As a data scientist, one should always look for accurate yet fast methods or functions to do the data modeling work. If the method is inherently slow, then it will create an execution bottleneck for large data sets.

A good way to determine scalability is to run the models for increasing data set size, extract the execution times for all the runs, and plot the trend.

Here is the boiler plate code for this. And here is the result. Due to their simplicity, stats.linregress and simple matrix inverse methods perform the fastest, even up to 10 million data points.



Summary

In this article, we discussed 7 effective ways to perform simple linear regression. Most of them are scalable to more generalized multi-variate and polynomial regression modeling too. We did not list the R^2 fit for these methods as all of them are very close to 1.

For a single-variable regression, with millions of artificially generated data points, the regression coefficient is estimated very well.

The goal of this article is primarily to discuss the relative speed/computational complexity of these methods. We showed the computational complexity measure of each of them by testing on a synthesized data set of increasing size (up to 10 million samples). Surprisingly, the simple matrix inverse analytic solution works pretty fast compared to scikit-learn's widely used Linear Model.

Author **Tirthajyoti Sarkar**

Dr. Tirthajyoti Sarkar, Senior Principal Engineer at ON Semiconductor, designs advanced power semiconductor technology and products. He also loves to experiment with advanced data analytics, machine learning, and Python/R programming. He writes for multiple Data Science/Artificial intelligence focused publications and loves to explore machine learning techniques for semiconductor designs. Tirthajyoti holds a Ph.D. from University of Illinois and is a Senior member of IEEE. He recently developed 'Pydbgen', a lightweight, open-source Python library for generating random database entities for software test engineering.