Demonstrations of Linear-Regressions in Python with Least Squares Method*

Daniel E. Hernández Zelada
Computer Science Student. (of FCE)
Universidad Francisco Marroquín
Guatemala, Guatemala
danielernesto@ufm.edu

***Abstract*—This document is a demonstration of Linear-Regressions through python. It solves two problems with two subsets of problems one of which is optimizing the best-fitting function to find the maximum value possible given some real world restrictions.**

***Index Terms*—linear algebra, regression, optimizing, least squares**

I. Introduction

This document explains the process of finding the least squares for linear-regressions on Python and using them ro predict data and maximize functions.

II. Least Squares

A. Description of the Problem

Often in the world, one finds that particular data gathered through time–for example–can be related with a linear equation. Least squares is a method of finding such linear equation through matrices constructed with data sets to build a best-fit function to approximate as accurately as possible to the real data.

These functions can be used to predict the behavior of events in the future. Though it's important to note that their precision will, of course, depend on the magnitude of the error.

B. Normal Equation

To initialize the construction of such system, one needs to be familiarized with the main equation used with the *Least Squares* method, i.e. the *Normal Equation* which is as follows.

$$A^T A\vec{x} = A^T\vec{b} \tag{1}$$

Matrix A is composed of 1s in the first column while the second column is built with x_1 (i.e. the independent variable(s)). Note: If the experiment had more than one x e.g. from x_1 to x_n it would still be constructed by the same principle, first column of 1s and the next columns would be the different independent variables. It can be of shape (m,n).

The $\vec{b}$ on the other hand is constructed with de dependent variables, often y it can only be of shape (m,1).

Thanks to Christiaan Ketelaar and Carlos Morales for their help on understanding linear algebra and its applications.

$$\begin{bmatrix} 1 & ax_1 & \dots & ex_n \\ 1 & bx_1 & \dots & fx_n \\ 1 & cx_1 & \dots & gx_n \\ \vdots & \vdots & \vdots & \vdots \\ 1 & dx_1 & \dots & hx_n \end{bmatrix}$$

(a) Matrix A.

$$\begin{bmatrix} 1 & 5 \\ 1 & 3 \\ 1 & 8 \\ 1 & 7 \\ 1 & 10 \end{bmatrix}$$

(b) Matrix A.

$$\begin{bmatrix} 4 \\ 5 \\ 5 \\ 5 \\ 6 \end{bmatrix}$$

(a) $\vec{b}$.

$$\begin{bmatrix} \frac{\sqrt{3}}{2} \\ \pi \\ 10.2 \\ 64 \\ 8 \end{bmatrix}$$

(b) $\vec{b}$.

Fig. 2: Examples of A Matrices

C. General Proceedings

After the equations have been completed, one needs to make an augmented matrix of the form:

$$\left[A^T A \;\middle|\; A^T\vec{b} \right]$$

Fig. 3: Augmented Matrix.

Once Gauss-Jordan elimination has been done, the solutions of said augmented matrix will be C_0 which will be the intercept and C_1 which will be the slope of the linear-approximation function. So when one builds the function with the given coefficients it yields as follows:

$$y = C_0 + C_1 x \tag{2}$$

III. Procedures for Exercise One

The first exercise was executed using two methods of the Least Squares algorithm. In the former, the column of 1s was added manually as seen on:

```
year_of_birth = np.array([
    [1, 1920],
        [1, 1930],
        [1, 1940],
        [1, 1950],
        [1, 1960],
        [1, 1970],
        [1, 1980],
        [1, 1990]])
```

The more automated version is found on:

```
A = np.vstack([x, np.ones(len(x))]).T
```

which is used in latter and in the second exercise; said method appends a second column of 1s but now at the right of the original numbers instead of the original formula which would have them on the left. The reason of doing so is because it's really just a prerequisite by the rest of NumPy's functions.

The equation used on both methods is the Normal Equation (as seen on: 1) and the proceedings followed the augmented matrix (as seen on: 3) which is necessary to be able to then get C_0's and C_1's values, though the second method solved the matrix through `m, b = np.linalg.lstsq(A, y, rcond=None)[0]` which automatically returns the coefficients in an array-like form which then can later be manipulated.

Then the linear-approximation function (2) was built to predict, as the clause *a)* asked, a new life expectancy estimate based on the last events for the year 2000.

A. Default Matrices

$$\begin{bmatrix} 1 & 1920 \\ 1 & 1930 \\ 1 & 1940 \\ 1 & 1950 \\ 1 & 1960 \\ 1 & 1970 \\ 1 & 1980 \\ 1 & 1990 \end{bmatrix}$$

(a) Matrix A

$$\begin{bmatrix} 54.1 \\ 59.7 \\ 62.9 \\ 68.2 \\ 69.7 \\ 70.8 \\ 73.7 \\ 75.4 \end{bmatrix}$$

(b) $\vec{b}$.

Fig. 4: Matrices extracted from Ex. One's data set

B. Normal Equations

Following the normal equation (1) we get that $A^T A$ is as follows:

$$\begin{bmatrix} 8 & 15640 \\ 15640 & 30580400 \end{bmatrix}$$

Fig. 5: First part of the Normal Equation

And we get $A^T\vec{b}$ as follows:

$$\begin{bmatrix} 534.5 \\ 1046169.0 \end{bmatrix}$$

Fig. 6: Second part of the Normal Equation

We now construct the augmented matrix and solve it by using Gauss-Jordan elimination:

$$\left[\begin{array}{cc|c} 8 & 15640 & 534.5 \\ 15640 & 30580400 & 1046169.0 \end{array}\right]$$

Fig. 7: Augmented Matrix.

Once solved we get that $C_0 = -501.77$ and $C_1 = 0.29083$. Thus constructing the best-fit linear approximation of:

$$y = -501.767 + 0.291x \qquad (3)$$

Now that we have the final function we can make a reasonably well prediction of the life expectancy of the year 2000. Given that the years are the independent variable we just do $f(2000) = -501.767 + 0.291(2000)$ which results in: 79.9 years.

C. Graph and Precision Analysis

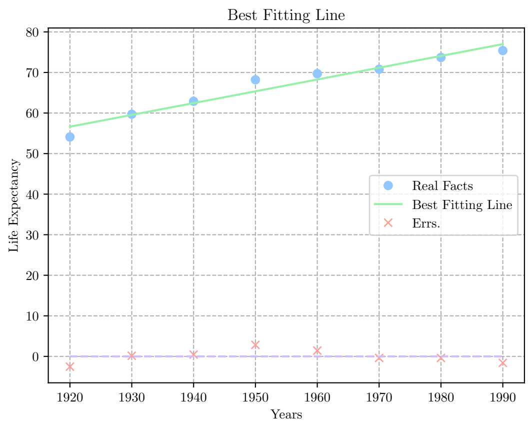

Fig. 8: Life expectancy in relation to decades and errors.

Given that the linear approximation fits sufficiently well with the data given, it is reasonable to conclude that this model is safe for making short-term predictions.

IV. Procedures for Exercise Two

The second exercise was built using only the automated method mentioned in the last procedures. However the data set was not 'hard-coded' but depends on an external file, an Excel file.

This exercise prompts the user for the file destination through the standard OS GUI for file exploring. Upon selecting the designated Excel file, the program loads the data frame

$$\begin{bmatrix} 0.59 & 1.0 \\ 0.8 & 1.0 \\ 0.95 & 1.0 \\ 0.45 & 1.0 \\ 0.79 & 1.0 \\ 0.99 & 1.0 \\ 0.9 & 1.0 \\ 0.65 & 1.0 \\ 0.79 & 1.0 \\ 0.69 & 1.0 \\ 0.79 & 1.0 \end{bmatrix}$$

(a) Matrix A Part 1

$$\begin{bmatrix} 0.49 & 1.0 \\ 1.09 & 1.0 \\ 0.95 & 1.0 \\ 0.79 & 1.0 \\ 0.65 & 1.0 \\ 0.45 & 1.0 \\ 0.6 & 1.0 \\ 0.89 & 1.0 \\ 0.79 & 1.0 \\ 0.99 & 1.0 \\ 0.85 & 1.0 \end{bmatrix}$$

(b) Matrix A Part 2

$$\begin{bmatrix} 3980 \\ 2200 \\ 1850 \\ 6100 \\ 2100 \\ 1700 \\ 2000 \\ 4200 \\ 2440 \\ 2300 \end{bmatrix}$$

(c) $\vec{b}$ Part 1

$$\begin{bmatrix} 6000 \\ 1190 \\ 1960 \\ 2760 \\ 4330 \\ 6960 \\ 4160 \\ 1990 \\ 2860 \\ 1920 \\ 2160 \end{bmatrix}$$

(d) $\vec{b}$ Part 2

Fig. 9: Matrix A and $\vec{b}$

into an array in the memory. Given that said array is of shape (22,2), the program extracts each side of the original array into two (22,1) numpy matrices as seen on:

```
import_file_path =
  ↪  tk.filedialog.askopenfilename()
cities_data =
  ↪  pd.read_excel(import_file_path)
processed_data =
  ↪  cities_data.to_numpy()
x_processed_data =
  ↪  np.array(processed_data[:, [0]])
y_processed_data =
  ↪  np.array(processed_data[:, [1]])
```

This results in Matrix A being:

Then it calculates–through numpy's linear algebra library–the coefficients of the best-fit approximation line as seen on:

```
m_2, b_2 = np.linalg.lstsq(A_2,
  ↪  y_processed_data, rcond=None)[0]
```

However this function will return two sets of matrices, so we extract the values inside of them (one for each matrix) and reassign `m_2` and `b_2` to those values, as seen on:

```
m_2 = m_2[0]
b_2 = b_2[0]
```

And that gives: $C_0 = 9510.096$ and $C_1 = -813.3645$ resulting on the following best-fit approximation line:

$$y = 9510.096 - 813.3645p \tag{4}$$

A. *Graph Analysis*

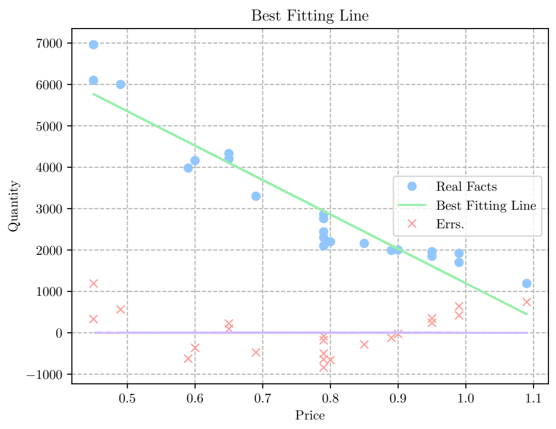

Fig. 10: Quantity in relation to prices and errors.

The approximated demand curve is enough to make fast decisions and to give a rough idea of how does the market moves with different prices but it's not recommended to use it alone for predictions, still more data is needed.

B. *Maximizing with Unitary Cost*

Now, given the unitary cost of \$0.23, it necessary to optimize the function through maximizing its profits (by playing with the different prices in the demand curve). To do this we first need to multiply our demand curve as follows:

$$y = (9510.096 - 813.3645p)(p - 0.23) \tag{5}$$

Then it needs to be differentiated and needs to be solved for p to find its roots and thus maximizing the function as shown in:

```
expresion__ = (b_2 + m_2 * p)*(p -
  ↪  0.23)
derivative = expresion__.diff(p)
print("Derivative: ")
pp(derivative)
solution = sp.solve(derivative, p)
```

The derivative of the function (5) is:

$$y = 11422.403 - 16628.73p \tag{6}$$

Thus it follows that `solve` has a value of 0.687 which means that the company could raise its price to a maximum of \$0.687 to optimize its profits.

Acknowledgment

I want to personally thank my Differential, Integral and Multivariate Calculus, and now Linear Algebra professor, Christiaan F. J. Ketelaar for not only for inspiring me but for encouraging me every day to give more of myself and for turning the classes into more dynamic ones with subjects of interest to someone who is studying computer science. I also want to give thanks to my parents for always supporting me through everything and for encouraging me through everything. Last but not least I want to thank God for giving me the opportunity of studying where I am in Guatemala and giving me the strength and wisdom to code everything that needed to be code.