Programming Exercise 1: Linear Regression

Thiago Raulino Dal Pont
INE6116000 - Conexionist Artificial Intelligence
Federal University of Santa Catarina

May 3, 2022

1 Introduction

In this first programming exercise, we are going to focus on Linear regression with one or more input variables.

Based on the Python Notebook available in Moodle, we implement some core functions for training Linear Regression models based on Gradient Descent and the Normal Equation method.

This work is divided as follows:

- Section 2 presents the required materials, methods and studies to finish the exercise;
- Section 3 presents the implementations, results and discussions;
- Section 4 presents the conclusions.

2 Methods and Materials

Before diving in the code, the exercise required some previous study in the Python libraries: `numpy` and `matplotlib`. To fulfill this goal, the documentations [1, 2] and some Youtube videos [3, 4] were used. A study on the Linear Regression using the reference book [5] was also required.

Altough model evaluation is not required by the exercise, we applied two metrics to estimate how well the models fit the train data:

- Root Mean Square Error (RMSE):

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_{act} - y_{pred})^2}{n}} \tag{1}$$

- R² - Coefficient of Determination

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_{act} - y_{pred})^2}{\sum_{i=1}^{n}(y_{act} - \bar{y})^2} \tag{2}$$

In terms of tools versions, we used:

- Python 3.9.7
- Numpy 1.20.3
- Matplotlib 3.5.1

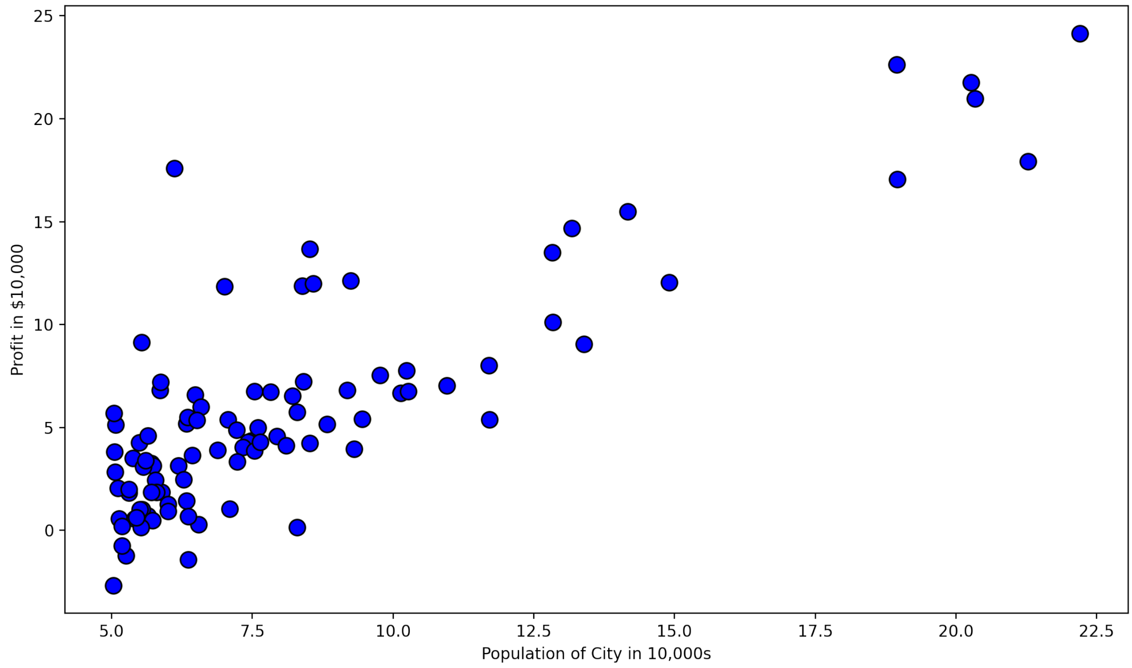
3 Results and Discussion

3.1 Linear regression with one variable

In this problem, we try to predict the profit of a food truck based on the city population.

The first step required in this section is the data visualization. Therefore, we implement the data plot using `pyplot` from `matplotlib` library, as shown in Figure 1.

Figure 1: Scatter plot of Population



The data shows a linear trend, which make its suitable for Linear Regression, according to Equation.

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{3}$$

The next task was to compute the cost function $J(\theta)$. Using `numpy` functions, we could easily apply linear algebra operations on matrices.

The code is presented below.

Listing 1: Cost Function implementation

```
import numpy as np

def computeCost(X, y, theta):

    # initialize some useful values
    m = y.size  # number of training examples

    # You need to return the following variables correctly
    J = 0

    # ======== YOUR CODE HERE ========
    ho_x = np.matmul(X, theta)            # Calculate ho(x) for all elements in X
    diff_mat = ho_x - y                   # Calculate the diff between all ho(x) and y
    diff_mat_square = diff_mat ** 2       # Square the elements in the diff array
    sum_op =  np.sum(diff_mat_square)     # Sum the elements of the squared array
    J = (1 / (2 * m)) * sum_op            # Multiply the sum with the constant
    # ==========================

    return J
```

Using the cost function, we implemented the Gradient Descent function

Listing 2: Cost Function implementation

```
def gradientDescent(X, y, theta, alpha, num_iters):
    # Initialize some useful values
    m = y.shape[0]  # number of training examples

    # make a copy of theta, to avoid changing the original array, since numpy arrays
    # are passed by reference to functions
    theta = theta.copy()

    J_history = []  # Use a python list to save cost in every iteration
```

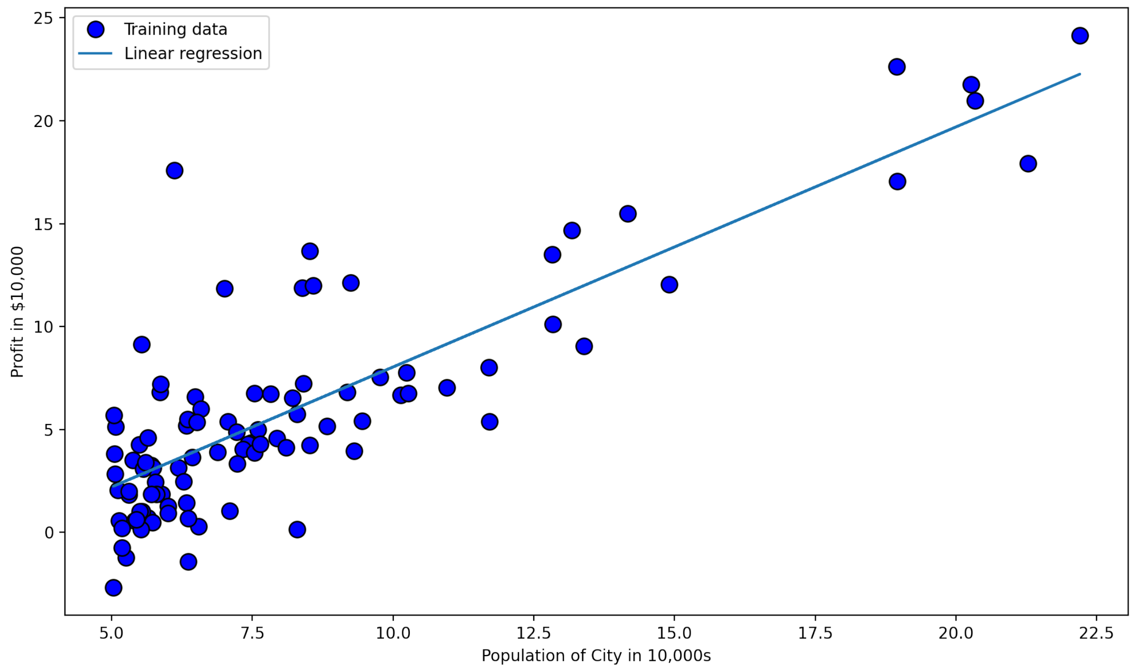
```
    for i in range(num_iters):
        # =========== YOUR CODE HERE =============
        ho_x = X @ theta                # [shape: (m,1)] Calculate ho(x) for all elements in X
        diff_mat = ho_x - y             # [shape: (m,1)] Calculate the diff between all ho(x) and y
        mult_diff_xj = diff_mat @ X     # [shape: (2,1)] Dot prod of diff with X matrix
        const_mult = alpha * (1/m)      # [shape: float] Get constant
        theta = theta - const_mult * mult_diff_xj # [shape: [2,1]]: Update theta values

        # ===========

        # save the cost J in every iteration
        J_history.append(computeCost(X, y, theta))

    return theta, J_history
```

After training the GD using $\alpha = 0.01$ with 1500 iterations, we obtained the following Equation:

$$h_\theta(x) = -3.6303 + 1.1664x \tag{4}$$

When plotting the hypothesis in Figure 2, we can see how well it fits the training data.

Figure 2: Plot hypothesis with training data



When calculating the regression metrics we obtained a RMSE of 2.99 and a R^2 of 0.70.

One interesting exercise is to plot the cost function surface according to θ_0 and θ_1 from the Equation 3

3.2 Linear Regression with Multiple Variables

In the section, we try to predict house prices using the area (ft^2) and the bedroom count.

Considering that the variables have distinct domain values, i.e., area is represented in thousands while bedroom in units, we need to normalize the dataset.

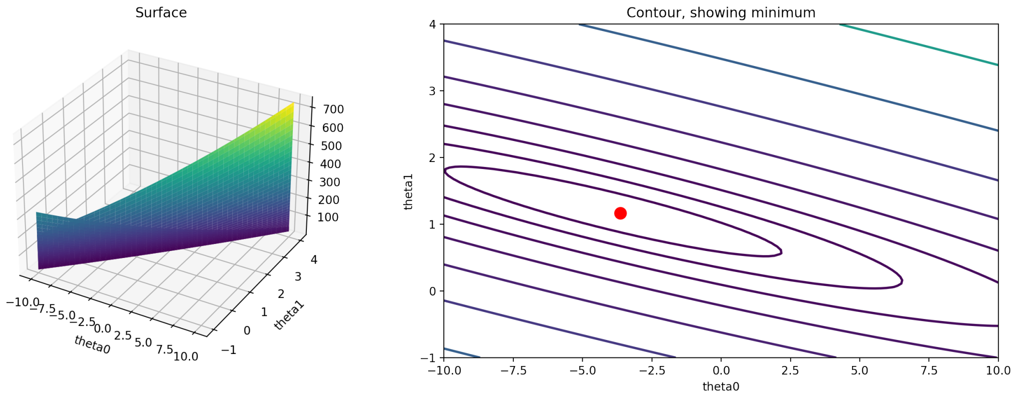
In the notebook, we applied the Standard Scaler, which represents the data as the z-score, using the formula:

$$X_{new} = \frac{X - \mu}{\sigma} \tag{5}$$

where μ is the mean of X and σ the standard deviation.

The code that implements the normalization is presented in Listing 3.

Figure 3: Cost function surface



Listing 3: Feature Normalization

```python
import numpy as np
def featureNormalize(X):

    X_norm = X.copy()
    # ================== YOUR CODE HERE ===========
    mu = np.mean(X_norm, axis=0)
    sigma = np.std(X_norm, axis=0)

    for i in range(len(mu)):
        X_norm[:, i] = X_norm[:, i] - mu[i]
        X_norm[:, i] = X_norm[:, i] / sigma[i]
    # ===================================
    return X_norm, mu, sigma
```

After implementing normalization, we adapted the cost function to allow multiple variable regression.

Listing 4: Cost function for LR with multiple variables

```python
def computeCostMulti(X, y, theta):

    # Initialize some useful values
    m = y.shape[0]  # number of training examples

    # ================= YOUR CODE HERE =============
    ho_x = np.matmul(X, theta)                  # Calculate ho(x) for all elements in X
    diff_mat = ho_x - y                         # Calculate the diff between all ho(x) and y
    diff_mat_square = np.dot(diff_mat.T, diff_mat)         # Square the elements in the diff array
    sum_op =  np.sum(diff_mat_square)           # Sum the elements of the squared array
    J = (1 / (2 * m)) * diff_mat_square         # Multiply the sum with the constant
    # ===================================

    return J
```

Finally, the GD algorithm for multi variables is presented in Listing 5.

Listing 5: GD for multi variable linear regression

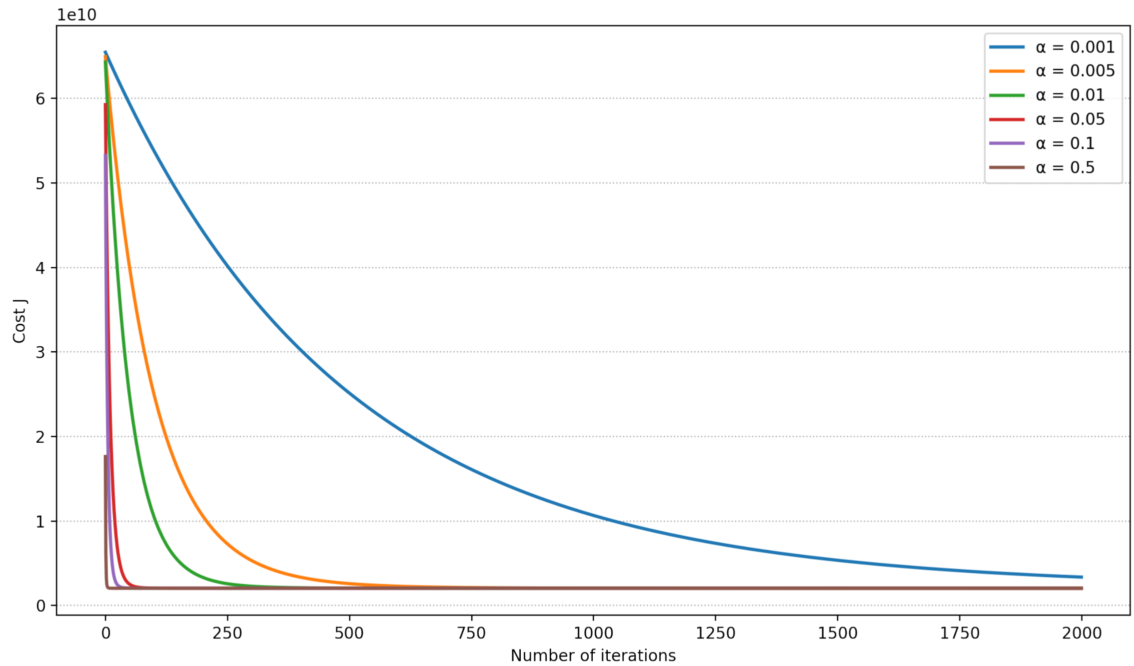
```python
def gradientDescentMulti(X, y, theta, alpha, num_iters):

    # Initialize some useful values
    m = y.shape[0]  # number of training examples

    # make a copy of theta, which will be updated by gradient descent
    theta = theta.copy()

    J_history = []

    for i in range(num_iters):
        # ==================== YOUR CODE HERE ================================
        ho_x = X @ theta                # [shape: (m,1)] Calculate ho(x) for all elements in X
        diff_mat = ho_x - y             # [shape: (m,1)] Calculate the diff between all ho(x) and y
```

```python
        mult_diff_xj = diff_mat @ X    # [shape: (2,1)] Dot prod of diff with X matrix
        const_mult = alpha * (1/m)     # [shape: float] Get constant
        theta = theta - const_mult * mult_diff_xj # [shape: [2,1]]: Update theta values
        # ==================================================================

        #print(X.shape, y.shape, ho_x.shape, diff_mat.shape, mult_diff_xj.shape, theta.shape)
        #print(theta)

        J = computeCostMulti(X, y, theta)
        # save the cost J in every iteration
        J_history.append(J)

        #if i % 100 == 0:
        #     print("Passo: %d;\tJ=%.2f;\ttheta=%s" %(i, J, str(theta)))

        #time.sleep(1)


    return theta, J_history
```

Although the algorithm may work with several learning rates (α), we may improve the results as we search for the optimal value.

We tested several values for learning rate, as shown in Figure 4.

Figure 4: GD using several α values



We may see GD may need from around 100 to more than 2000 iterations to converge depending on the learning rate.

If we consider both the number of iterations required for the convergence and the cost, we may estimate that a value of $\alpha = 0.05$ is a good choice.

Table 1 presents the metrics after 2000 iterations for some values of learning rates.

3.3 Normal Equations

The values of θ_0 and θ_1 may be acquired analytically, using the following equation.

$$\theta = (X^T X)^{-1} X^T y \tag{6}$$

Using numpy library, such equation can be easily implemented as shown in Listing 6.

Table 1: Regression metrics for some values of learning rate

α	RMSE	R^2
0.001	82037	0.71
0.005	63926	0.73
0.010	63926	0.73
0.050	63926	0.73
0.100	63926	0.73
0.500	63926	0.73

Listing 6: Normal Equation for multi variable linear regression

```
def normalEqn(X, y):
    theta = np.zeros(X.shape[1])

    # ===================== YOUR CODE HERE ============================
    x_t = X.T
    xt_x = np.matmul(x_t, X)
    xt_x_inv = np.linalg.inv(xt_x)
    xt_x_inv_xt = np.matmul(xt_x_inv, x_t)
    theta = np.matmul(xt_x_inv_xt, y)
    # =================================================================
    return theta
```

Using the normal equation, we achieved a RMSE of 63926, a R^2 of 0.73. It is worth reminding that those are the same metrics achieved by the Gradient Descent algorithm, although the values of θ[1] are quite different.

4 Final Remarks

In this exercise, we focused on implementing the Linear Regression using Gradient Descent and the Normal Equation, with one and multiple variables.

Using the `numpy` library, applying linear algebra operations with matrices was straight forward.

In terms of values of α, we saw that a low value makes it difficult for the algorithms to converge, while higher values may reduce the number of iterations required for convergence. However, although not presented in this exercise, even higher values of α may blow up the the cost function values.

References

[1] "Mathematical functions - numpy." https://numpy.org/doc/stable/reference/routines.math.html. Accessed 30 Apr 2022.

[2] "Matplotlib 3.5.1 documentation." https://matplotlib.org/3.5.1/index.html. Accessed 30 Apr 2022.

[3] "Complete python numpy tutorial (creating arrays, indexing, math, statistics, reshaping)." https://www.youtube.com/watch?v=GB9ByFAIAH4. Accessed 30 Apr 2022.

[4] "Intro to data visualization in python with matplotlib! (line graph, bar chart, title, labels, size)." https://www.youtube.com/watch?v=DAQNHzOcO5A. Accessed 30 Apr 2022.

[5] S. Haykin, *Neural networks and Learning Machines*. 2008.

[1] See details in the notebook