# Regressão Linear

**Arley dos Santos Ribeiro**
Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais
Belo Horizonte, Av. Amazonas 7675
arley.sribeiro@gmail.com

## Abstract

Este trabalho tem como objetivo implementar o método de Regressão Linear Simples e a Regressão Linear Múltipla. E utilizar esse algoritmo em um banco de dados real, para verificar a aproximação linear dos dados.

## 1 Regressão Linear Simples

A regressão linear é um método para estimar o valor de uma variável $y$, com o objetivo verificar a existência de uma relação funcional entre uma variável dependente $y$ com uma ou mais variáveis independentes $x$. Com isso, podemos estimar um valor (predição) para a relação de variáveis observadas. Para iniciar o processo de modelagem deve-se observar se há uma relação entre os termos de interesse, podemos usar um diagrama de dispersão para determinar se duas variáveis tem uma relação. A regressão linear têm uma equação da forma abaixo:

$$Y_i = \alpha + \beta X_i + \epsilon_i \tag{1}$$

Dessa relação temos que:

- $Y_i$ **:** É a variável dependente que o terá o valor aproximado.
- $\alpha$ **:** É uma constante, que representa a interceptação da reta com o eixo vertical.
- $\beta$ **:** É outra constante, que representa o coeficiente angular da reta.
- $X_i$ **:** É a variável independente.
- $\epsilon$ **:** é o erro que está associado à distância entre o valor observado $Y_i$ e o correspondente ponto na curva, do modelo proposto, para o mesmo nível de $X_i$.

### 1.1 Método dos Mínimos Quadrados

Para minimizar o erro de medição é utilizado o método do mínimos quadrados, este calcula o melhor ajuste para os dados observados, minimizando a soma dos quadrados dos desvios verticais de cada ponto para a linha (se um ponto estiver exatamente na linha ajustada, então seu desvio vertical é 0). Como os desvios são quadráticos, não há o cancelamento entre valores positivos e negativos.

$$\epsilon_i = Y_i - \alpha - \beta X_i \tag{2}$$

Elevando ambos os lados da equação ao quadrado,

$$\epsilon_i^2 = (Y_i - \alpha - \beta X_i)^2 \tag{3}$$

Aplicando o somatório,

$$\sum_{i=1}^{n} \epsilon_i^2 = \sum_{i=1}^{n} (Y_i - \alpha - \beta X_i)^2 \tag{4}$$

Após a regressão um ou mais pontos podem ficar longe da linha (e, portanto tem um grande valor residual), este ponto é conhecido como um outlier. Que podem representar dados errados, ou indicam um regressão mal ajustada. Pontos que estão muito longe da reta podem influenciar na sua inclinação.

## 2 Gradiente descendente

O gradiente descendente é um método numérico usado em otimização, para encontrar um mínimo (local) de uma função, usando um algoritmo iterativo, onde cada passo se torna a direção do gradiente, que corresponde à direção de declive máximo.

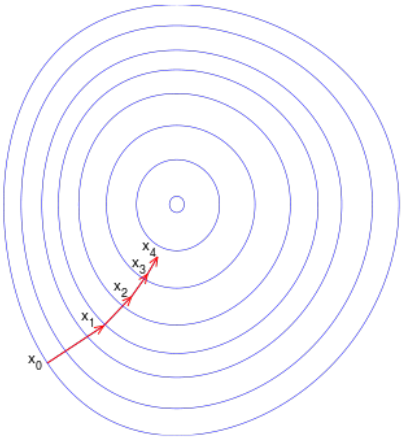


Figura 1: Direção do Gradiente

### 2.1 Equação de Atualização

Para o calculo do gradiente, precisamos de uma função para minimizar o erro da predição para um conjunto de pontos. Assim, o melhor ajuste para minimizar o erro, é a soma dos quadrados das diferenças entre o valor estimado e os dados observados, estas diferenças são chamadas de resíduos.

Com isso, o objetivo da regressão linear é minimizar o custo da função:

$$J(\beta) = \frac{1}{2m} \sum_{i=1}^{n} (h_\beta(x_i - y_i))^2 \tag{5}$$

onde, a $h_\beta$ é dado por:

$$h_\beta(x) = \beta^T x = \beta_0 + \beta_1 x_1 \tag{6}$$

#### 2.1.1 Tarefa 1: Implementar a função de custo

A função $compute\_cost(X, y, beta)$, têm como objetivo calcular o melhor ajuste para o conjunto de dados procurando minimizar a soma dos quadrados das diferenças entre o valor estimado e os dados observados, estas diferenças são chamadas de resíduos. Utilizando as equações 5 e 6.

$$J(\beta) = \frac{1}{2m}\sum_{i=1}^{n}(h_\beta(x_i - y_i))^2 \tag{7}$$

$$h_\beta(x) = \beta^T x = \beta_0 + \beta_1 x_1 \tag{8}$$

Para auxiliar a visualização, podemos reescrever $X$, $y$ e $\beta$ na forma matricial:

$$y = \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{pmatrix}, X = \begin{pmatrix} x_{11}^T & ... & x_{1m}^T \\ x_{21}^T & . & x_{2m}^T \\ . & . & . \\ . & . & . \\ . & . & . \\ x_{n1}^T & ... & x_{nm}^T \end{pmatrix}, \beta = \begin{pmatrix} \beta_1 & \beta_2 & ... & \beta_n \end{pmatrix}, \tag{9}$$

Assim teriamos $(h_\beta(x_i - y_i))^2$ como:

$$\begin{pmatrix} X * \beta^T - y \end{pmatrix}^2 = \left( \begin{pmatrix} x_{11}^T & ... & x_{1m}^T \\ x_{21}^T & . & x_{2m}^T \\ . & . & . \\ . & . & . \\ . & . & . \\ x_{n1}^T & ... & x_{nm}^T \end{pmatrix} * \begin{pmatrix} \beta_1 \\ \beta_2 \\ . \\ . \\ . \\ \beta_n \end{pmatrix} - \begin{pmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{pmatrix} \right)^2 \tag{10}$$

Dessa forma, fica fácil visualizar um código em python para a equação 10, utilizando as seguintes rotinas:

- `.dot`: É equivalente a multiplicação matricial.
- `np.power`: Cada posição do array é elevado ao valor do segundo parâmetro da função.
- `beta.T`: Retorna a matriz transposta.
- `np.sum`: Soma os elementos da matriz em um determinado eixo.

O código dessa função encontra-se no apêndice 4.5 na página 17.

### 2.2 Calculando o gradiente descendente

- A descida do gradiente determina um vetor de pesos que minimiza E, começando com um vetor inicial de pesos arbitrário e modificando-o repetidamente em pequenos passos ($\alpha$);
- A cada passo. o vetor de pesos é alterado na direão que produz a maior queda ao longo da superfície de erro.
- O processo contínua até convergir para o mínimo.

$$\beta_j = \beta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}(h_\beta(x_i) - y_i)x_{ij} \tag{11}$$

#### 2.2.1 Tarefa 2: Implementar a Eq. 11

Devemos observar que essa função calcula e atualiza $\beta_j$ simultaneamente, portanto, é necessário dividir esse somatório em partes.

$$\beta_j = \beta_j - \alpha\frac{1}{n}\sum_{i=1}^{n}(h_\beta(x_i) - y_i)x_{ij} \tag{12}$$

1. A primeira ficará encarregada de calcular essa parte $(h_\beta(x_i) - y_i)$ chamaremos essa parte de `resíduos` que será um array com valores para cada ponto.

2. A segunda parte ficará encarregada de multiplicar todos os `resíduos` por $X_{ij}$, que chamaremos de `gradient`.
3. A terceira parte será somar todos os valores do `gradient` e multiplicar por $(\alpha/n)$ e armazenar em uma variável temporária.
4. A quarta parte consiste em atualizar $\beta_j$.

O código dessa função encontra-se no apêndice A na página 17.

# 3 Machine Learning Exercise 1 - Linear Regression

## 3.1 Linear regression with one variable

```
In [1]: import numpy as np
        import pandas as pd
        import os
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D
        from sklearn import datasets, linear_model

        %matplotlib inline
```

```
In [2]: data = pd.read_csv('ex1data1.txt', header=None, names=['Population', 'Profit'
        data.head()
```

```
Out[2]:    Population   Profit
        0      6.1101  17.5920
        1      5.5277   9.1302
        2      8.5186  13.6620
        3      7.0032  11.8540
        4      5.8598   6.8233
```

```
In [3]: data.describe()
```

```
Out[3]:         Population     Profit
        count   97.000000  97.000000
        mean     8.159800   5.839135
        std      3.869884   5.510262
        min      5.026900  -2.680700
        25%      5.707700   1.986900
        50%      6.589400   4.562300
        75%      8.578100   7.046700
        max     22.203000  24.147000
```

```
In [4]: ax = data.plot(kind='scatter', x='Population', y='Profit', title='Scatter plo
        ax.set_xlabel('Population of city in 10,000s')
        ax.set_ylabel('Profit in $10,000s')
```

```
Out[4]: <matplotlib.text.Text at 0x7fecdb5e7310>
```

max size=0.90.9ex1$_files/ex1_{51}.png$

## 3.2 Gradient Descent

First, you create a function to compute the cost of a given solution (characterized by the parameters beta):

```
In [5]: def compute_cost(X, y, beta):
            m = len(X)
            hb = np.power((X.dot(beta.T) - y),2)#multiplication matrix .dot
            return np.sum(hb) / (2 * m)
```

We store each example as a row in the X matrix. To take into account the intercept term ($\beta 0$), we add an additional first column to X and set it to all ones. This allows us to treat$\beta 0$ as simply another 'feature'.

```
In [6]: data.insert(0, 'beta zero', 1)
```

Now let's do some variable initialization

```
In [7]: # set X (training data) and y (target variable)
        cols = data.shape[1]
        X = data.iloc[:,0:cols-1]
        y = data.iloc[:,cols-1:cols]
```

Now, you need to guarantee that X (training set) and y (target variable) are correct.

```
In [8]: X.head()
```

```
Out[8]:    beta zero  Population
        0          1      6.1101
        1          1      5.5277
        2          1      8.5186
        3          1      7.0032
        4          1      5.8598
```

```
In [9]: y.head()
```

```
Out[9]:     Profit
        0  17.5920
        1   9.1302
        2  13.6620
        3  11.8540
        4   6.8233
```

The cost function is expecting numpy matrices so we need to convert X and y before we can use them. We also need to initialize beta.

```
In [10]: X = np.matrix(X.values)
         y = np.matrix(y.values)
         beta = np.matrix(np.array([0,0]))
```

Here's what beta looks like.

```
In [11]: beta
```

```
Out[11]: matrix([[0, 0]])
```

Let's take a quick look at the shape of our matrices.

```
In [12]: X.shape, beta.shape, y.shape
```

```
Out[12]: ((97, 2), (1, 2), (97, 1))
```

Now let's compute the cost for our initial solution (0 values for beta).

```
In [13]: compute_cost(X, y, beta)
```

```
Out[13]: 32.072733877455676
```

Now, you are asked to define a function to perform gradient descent on the parameters beta

```
In [14]: def gradient_descent(X, y, theta, alpha, iters):
             '''
             alpha: learning rate
             iters: number of iterations
             OUTPUT:
```

```
    theta: learned parameters
    cost:  a vector with the cost at each training iteration
    '''
    temp       = np.matrix(np.zeros(theta.shape)) #np.zero: Return a new ar
    parameters = int(theta.ravel().shape[1]) #Return a contiguous flattened
    cost       = np.zeros(iters)
    n = len(X)
    for i in range(iters):
        for j in range(parameters):
            gradient = np.multiply((X.dot(theta.T) - y),X[:,j])
            temp[0,j] -= (alpha/(n)) * np.sum(gradient)

        theta = temp #update
        cost[i] = compute_cost(X, y, theta)
    return theta, cost
```

Initialize some additional variables - the learning rate alpha, and the number of iterations to perform

```
In [15]: alpha = 0.01
         iters = 1500
```

Now let's run the gradient descent algorithm to fit our parameters theta to the training set.

```
In [16]: g, cost = gradient_descent(X, y, beta, alpha, iters)
         print ("Coeficiente angular da função")
         print (g)
         print ("\nResiduos")
         print cost
```

```
Coeficiente angular da função
[[-3.63609474  1.16699229]]

Residuos
[ 6.73719046  5.93238265  5.90102959 ...,  4.48315549  4.48313304
  4.48311066]
```

Finally we can compute the cost (error) of the trained model using our fitted parameters.

```
In [17]: print ("Error do treino")
         compute_cost(X, y, g)
```

```
Error do treino
```

```
Out[17]: 4.483110660040591
```

Now let's plot the linear model along with the data to visually see how well it fits.

```
In [18]: x = np.linspace(data.Population.min(), data.Population.max(), 100)
         f = g[0, 0] + (g[0, 1] * x)

         fig, ax = plt.subplots(figsize=(8,4))
         ax.plot(x, f, 'r', label='Prediction')
         ax.scatter(data.Population, data.Profit, label='Traning Data')
         ax.legend(loc=2)
         ax.set_xlabel('Population of city in 10,000s')
         ax.set_ylabel('Profit in $10,000s')
         ax.set_title('Predicted Profit vs. Population Size')
         ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_3 3_0.png$

Looks pretty good! Remember that the gradient decent function also outputs a vector with the cost at each training iteration, we can plot it as well.

Since the cost always decreases - this is an example of a convex optimization problem.

```
In [19]: fig, ax = plt.subplots(figsize=(8,4))
         ax.plot(np.arange(iters), cost, 'r')
         ax.set_xlabel('Iterations')
         ax.set_ylabel('Cost')
         ax.set_ylim(4.0)
         ax.set_title('Error vs. Training Epoch')
         ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_3 5_0.png$

Now, we will show a contour plot that presents beta0 against beta1 and the outcome of J. First, we set values for beta0 and beta1

```
In [20]: beta0_vals = np.linspace(-10, 10, 100)
         beta1_vals = np.linspace(-1, 4, 100)
```

Now, initialize J values to a matrix of 0's

```
In [21]: j_vals = np.zeros([len(beta0_vals), len(beta1_vals)])
```

```
In [22]: for i in range(len(beta0_vals)):
             for j in range(len(beta1_vals)):
                 t = np.matrix(np.array([beta0_vals[i], beta1_vals[j]]))
                 j_vals[i,j] = compute_cost(X, y, t)
```

```
In [23]: plt.contour(beta0_vals, beta1_vals, j_vals.T, np.logspace(-2, 3, 20));
```

max size=0.90.9ex1$_f iles/ex1_4 1_0.png$

```
In [24]: plt.scatter(g[0,0],g[0,1],)
         plt.contour(beta0_vals, beta1_vals, j_vals.T, np.logspace(-2, 3, 20));
```

max size=0.90.9ex1$_f iles/ex1_4 2_0.png$

Now, in 3D

```
In [25]: beta0_mesh, beta1_mesh = np.meshgrid(beta0_vals, beta1_vals)
         fig = plt.figure()
         ax = fig.gca(projection='3d')
         ax.plot_surface(beta0_mesh, beta1_mesh, j_vals.T);
```

max size=0.90.9ex1$_f iles/ex1_4 4_0.png$

### 3.3 Linear regression with multiple variables

From now on, you will use the second dataset, i.e., ex1data2.txt. This is a housing price dataset with 2 variables (size of the house in square feet and number of bedrooms) and a target (price of the house). You are asked to use the techniques already applied to analyze that data set.

```
In [26]: data2 = pd.read_csv('ex1data2.txt', header=None, names=['Size', 'Bedrooms',
         data2.head()

Out[26]:    Size  Bedrooms   Price
         0  2104         3  399900
         1  1600         3  329900
         2  2400         3  369000
         3  1416         2  232000
         4  3000         4  539900
```

For this task we add another pre-processing step - normalizing the features.

Notice that the scale of the values for each feature is vastly large. A house will typically have 2-5 bedrooms, but may have anywhere from hundreds to thousands of square feet. If we use the features as they are in the dataset, the 'size' feature would too much wheighted and would end up dwarfing any contributions from the 'number of bedrooms' feature. To fix this, we need to do something called 'feature normalization'. That is, we need to adjust the scale of the features to level the playing field. One way to do this is by subtracting from each value in a feature the mean of that feature, and then dividing by the standard deviation.

```
In [27]: data2 = (data2 - data2.mean()) / data2.std()
         data2.head()

Out[27]:        Size  Bedrooms     Price
         0  0.130010 -0.223675  0.475747
         1 -0.504190 -0.223675 -0.084074
         2  0.502476 -0.223675  0.228626
         3 -0.735723 -1.537767 -0.867025
         4  1.257476  1.090417  1.595389
```

Given that you were asked to implement both cost function and gradient descent using matrix operations, your previously implementations will work just fine in the multivariate dataset. Hence, you need now insert the 'ones' column as before and separate the X's and the y's.

Conduct the rest of this exercise by repeating the experiments conducted in the simple linear dataset...

```
In [28]: data2.insert(0, 'Ones', 1)

In [29]: cols = data2.shape[1]
         X2 = data2.iloc[:,0:cols-1]
         y2 = data2.iloc[:,cols-1:cols]

In [30]: X2.head()

Out[30]:    Ones      Size  Bedrooms
         0     1  0.130010 -0.223675
         1     1 -0.504190 -0.223675
         2     1  0.502476 -0.223675
         3     1 -0.735723 -1.537767
         4     1  1.257476  1.090417

In [31]: y2.head()

Out[31]:       Price
         0  0.475747
         1 -0.084074
         2  0.228626
         3 -0.867025
         4  1.595389
```

```
In [32]: X2 = np.matrix(X2.values)
         y2 = np.matrix(y2.values)
         beta2 = np.matrix(np.array([0,0,0]))

In [33]: beta2

Out[33]: matrix([[0, 0, 0]])

In [34]: X2.shape, beta2.shape, y2.shape

Out[34]: ((47, 3), (1, 3), (47, 1))

In [35]: g2, cost2 = gradient_descent(X2, y2, beta2, alpha, iters)
         print ("Coeficiente angular da função")
         print (g2)
         print ("\nResiduos")
         print cost2

Coeficiente angular da função
[[ -9.99791266e-17   8.84056762e-01  -5.24711271e-02]]

Residuos
[ 0.4805491   0.47200469  0.46370085 ...,  0.1306867   0.1306867   0.1306867 ]

In [36]: print "Erro do treino"
         compute_cost(X2, y2, g2)

Erro do treino

Out[36]: 0.13068669669956798

In [37]: x = np.linspace(data2.Size.min(), data2.Size.max(), 100)
         f = g2[0, 0] + (g2[0, 1] * x) + (g2[0, 2] * x)

         fig, ax = plt.subplots(figsize=(8,4))
         ax.plot(x, f, 'r', label='Prediction')
         ax.scatter(data2.Size, data2.Price, label='Traning Data2')
         ax.legend(loc=2)
         ax.set_xlabel('Size')
         ax.set_ylabel('Price')
         ax.set_title('Predicted Price vs. Size')
         ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_6 2_0 .png$

```
In [38]: x = np.linspace(data2.Bedrooms.min(), data2.Bedrooms.max(), 100)
         f = g2[0, 0] + (g2[0, 1] * x) + (g2[0, 2] * x)

         fig, ax = plt.subplots(figsize=(8,4))
         ax.plot(x, f, 'r', label='Prediction')
         ax.scatter(data2.Bedrooms, data2.Price, label='Traning Data2')
         ax.legend(loc=2)
         ax.set_xlabel('Bedrooms')
         ax.set_ylabel('Price')
         ax.set_title('Predicted Price vs. Bedrooms')
         ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_6 3_0 .png$

```
In [39]: fig, ax = plt.subplots(figsize=(12,8))
         ax.plot(np.arange(iters), cost2, 'r')
         ax.set_xlabel('Iterations')
         ax.set_ylabel('Cost')
         ax.set_title('Error vs. Training Epoch')
         ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_6 4_0 .png$

```
In [40]: beta0_vals = np.linspace(-10, 10, 100)
         beta1_vals = np.linspace(-1, 4, 100)

In [41]: j_vals = np.zeros([len(beta0_vals), len(beta1_vals)])

In [42]: for i in range(len(beta0_vals)):
             for j in range(len(beta1_vals)):
                 t = np.matrix(np.array([beta0_vals[i], beta1_vals[j]]))
                 j_vals[i,j] = compute_cost(X, y, t)

In [43]: plt.contour(beta0_vals, beta1_vals, j_vals.T, np.logspace(-2, 3, 20));
```

max size=0.90.9ex1$_f iles/ex1_6 8_0 .png$

```
In [44]: plt.scatter(g2[0,0],g2[0,1],)
         plt.contour(beta0_vals, beta1_vals, j_vals.T, np.logspace(-2, 3, 20));
```

max size=0.90.9ex1$_f iles/ex1_6 9_0 .png$

```
In [45]: beta0_mesh, beta1_mesh = np.meshgrid(beta0_vals, beta1_vals)
         fig = plt.figure()
         ax = fig.gca(projection='3d')
         ax.plot_surface(beta0_mesh, beta1_mesh, j_vals.T);
```

max size=0.90.9ex1$_f iles/ex1_7 0_0 .png$

### 3.4 A real world dataset

```
In [46]: import graphlab
         graphlab.canvas.set_target('ipynb')
```

### 3.5 Load groupon data

```
In [47]: data3 = pd.read_csv('groupon-deals.csv')
         data3.head()
         cols = data3.shape[1]
         data4 =data3.iloc[:,2:cols-10]
         data4
```

Out[47]:

| | value | you_save | discount_pct | num_sold |
|---|---|---|---|---|
| 0 | 30 | 15 | 50 | 108 |
| 1 | 70 | 35 | 50 | 255 |
| 2 | 99 | 54 | 55 | 352 |
| 3 | 65 | 35 | 54 | 501 |
| 4 | 134 | 67 | 50 | 359 |
| 5 | 155 | 70 | 45 | 45 |

| | | | | |
|---|---|---|---|---|
| 6 | 150 | 75 | 50 | 210 |
| 7 | 20 | 10 | 50 | 163 |
| 8 | 10 | 5 | 50 | 823 |
| 9 | 50 | 25 | 50 | 3 |
| 10 | 135 | 76 | 56 | 545 |
| 11 | 40 | 22 | 55 | 95 |
| 12 | 10 | 5 | 50 | 4749 |
| 13 | 99 | 61 | 62 | 2074 |
| 14 | 160 | 91 | 57 | 145 |
| 15 | 50 | 25 | 50 | 595 |
| 16 | 189 | 100 | 53 | 179 |
| 17 | 20 | 11 | 55 | 243 |
| 18 | 55 | 35 | 64 | 99 |
| 19 | 23 | 11 | 48 | 3026 |
| 20 | 175 | 116 | 66 | 348 |
| 21 | 30 | 15 | 50 | 4233 |
| 22 | 194 | 134 | 69 | 21 |
| 23 | 130 | 81 | 62 | 214 |
| 24 | 17 | 9 | 53 | 922 |
| 25 | 15 | 8 | 53 | 1085 |
| 26 | 250 | 151 | 60 | 71 |
| 27 | 30 | 15 | 50 | 37 |
| 28 | 10 | 5 | 50 | 444 |
| 29 | 140 | 105 | 75 | 315 |
| ... | ... | ... | ... | ... |
| 16662 | 120 | 60 | 50 | 198 |
| 16663 | 25 | 13 | 52 | 594 |
| 16664 | 350 | 200 | 57 | 68 |
| 16665 | 50 | 25 | 50 | 1840 |
| 16666 | 120 | 60 | 50 | 730 |
| 16667 | 20 | 10 | 50 | 505 |
| 16668 | 750 | 565 | 75 | 725 |
| 16669 | 240 | 161 | 67 | 831 |
| 16670 | 35 | 20 | 57 | 1303 |
| 16671 | 104 | 65 | 63 | 361 |
| 16672 | 30 | 15 | 50 | 433 |
| 16673 | 100 | 55 | 55 | 321 |
| 16674 | 59 | 29 | 49 | 671 |
| 16675 | 47 | 32 | 68 | 4041 |
| 16676 | 36 | 18 | 50 | 636 |
| 16677 | 40 | 20 | 50 | 1506 |
| 16678 | 30 | 15 | 50 | 828 |
| 16679 | 199 | 129 | 65 | 96 |
| 16680 | 99 | 61 | 62 | 593 |
| 16681 | 190 | 95 | 50 | 189 |
| 16682 | 70 | 35 | 50 | 785 |
| 16683 | 219 | 120 | 55 | 32 |
| 16684 | 30 | 15 | 50 | 767 |
| 16685 | 50 | 35 | 70 | 138 |
| 16686 | 1040 | 852 | 82 | 321 |
| 16687 | 50 | 25 | 50 | 235 |
| 16688 | 120 | 71 | 59 | 178 |
| 16689 | 40 | 20 | 50 | 163 |
| 16690 | 209 | 130 | 62 | 61 |
| 16691 | 30 | 15 | 50 | 410 |

[16692 rows x 4 columns]

```
In [48]: data4 = (data4 - data4.mean()) / data4.std()
         data4.head()
```

Out[48]:

| | value | you_save | discount_pct | num_sold |
|---|---|---|---|---|
| 0 | -0.340996 | -0.354025 | -0.608165 | -0.353680 |
| 1 | -0.208339 | -0.249447 | -0.608165 | -0.263808 |
| 2 | -0.112163 | -0.150097 | -0.107899 | -0.204505 |
| 3 | -0.224921 | -0.249447 | -0.207952 | -0.113411 |
| 4 | 0.003912 | -0.082121 | -0.608165 | -0.200226 |

```
In [49]: data4.insert(0, 'Ones', 1)

In [50]: cols = data4.shape[1]
         X4 = data4.iloc[:,0:cols-1]
         y4 = data4.iloc[:,cols-1:cols]

In [51]: X4.head()
```

Out[51]:

| | Ones | value | you_save | discount_pct |
|---|---|---|---|---|
| 0 | 1 | -0.340996 | -0.354025 | -0.608165 |
| 1 | 1 | -0.208339 | -0.249447 | -0.608165 |
| 2 | 1 | -0.112163 | -0.150097 | -0.107899 |
| 3 | 1 | -0.224921 | -0.249447 | -0.207952 |
| 4 | 1 | 0.003912 | -0.082121 | -0.608165 |

```
In [52]: y4.head()
```

Out[52]:

| | num_sold |
|---|---|
| 0 | -0.353680 |
| 1 | -0.263808 |
| 2 | -0.204505 |
| 3 | -0.113411 |
| 4 | -0.200226 |

```
In [53]: X4 = np.matrix(X4.values)
         y4 = np.matrix(y4.values)
         beta4 = np.matrix(np.array([0,0,0,0]))

In [54]: g4, cost4 = gradient_descent(X4, y4, beta4, alpha, iters)
         print ("Coeficiente angular da função")
         print (g4)
         print ("\nResiduos")
         print cost4

Coeficiente angular da função
[[  2.37358393e-17  -9.09830831e-02   1.63258208e-02  -2.46783929e-02]]

Residuos
[ 0.49981988  0.49967694  0.49954005 ...,  0.49606231  0.49606215
  0.49606199]

In [55]: print "Erro do treino"
         compute_cost(X4, y4, g4)

Erro do treino

Out[55]: 0.49606199117546113

In [56]: x = np.linspace(data4.value.min(), data4.value.max(), 100)
         f = g4[0, 0] +  (g4[0, 1] * x) +  (g4[0, 2] * x) +  (g4[0, 3] * x)
```

```
fig, ax = plt.subplots(figsize=(8,4))
ax.plot(x, f, 'r', label='Prediction')
ax.scatter(data4.value, data4.num_sold, label='Traning Data4')
ax.legend(loc=2)
ax.set_xlabel('value')
ax.set_ylabel('num_sold')
ax.set_title('Predicted num_sold vs. value')
ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_8 3_0 .png$

```
In [57]: fig, ax = plt.subplots(figsize=(8,4))
         ax.plot(np.arange(iters), cost4, 'r')
         ax.set_xlabel('Iterations')
         ax.set_ylabel('Cost')
         ax.set_ylim(4.0)
         ax.set_title('Error vs. Training Epoch')
         ax.grid(True)
```

max size=0.90.9ex1$_f iles/ex1_8 4_0 .png$

# 4 Análise dos Resultados

Nesta seção iremos analisar o resultados que se encontram na seção 3.

## 4.1 Função Gradiente uma variável

Analisando os resultados obtidos ao simular a função gradiente figura 2

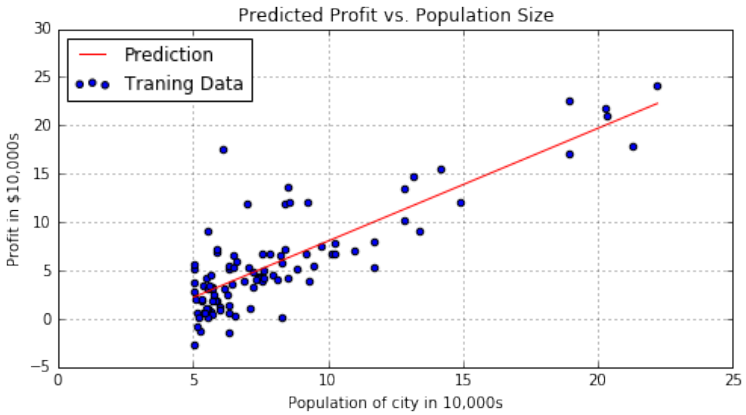


Figura 2: Linearização dos pontos

Podemos observa que o resultado foi o esperado, uma reta que aproxima bem os valores da população com o lucro.

## 4.2 Função Custo

Analisando os resultados obtidos ao simular a função gradiente, obtivemos um array com os custos a da iteração, figura 3

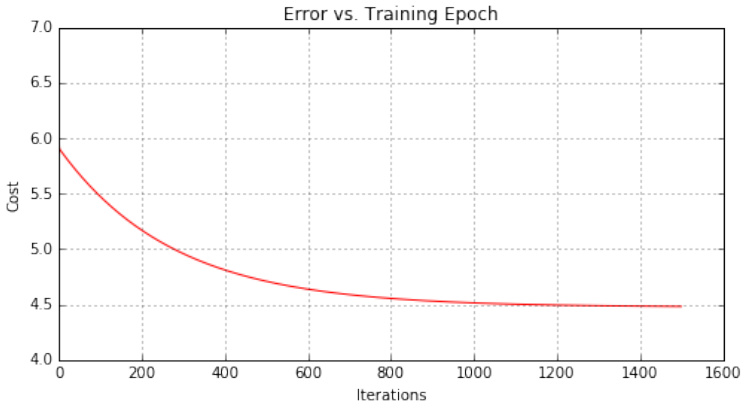


Figura 3: Linearização dos pontos

Percebemos que os custos a cada iteração diminui como o esperado, portanto concluímos que para a linearização simples o algoritmo foi satisfatório.

### 4.3 Função Gradiente para muitas variáveis

Analisando os resultados obtidos ao simular a função gradiente para mais de uma variável, figura 4, neste caso, o preço da casa está em função do tamanho do imóvel.

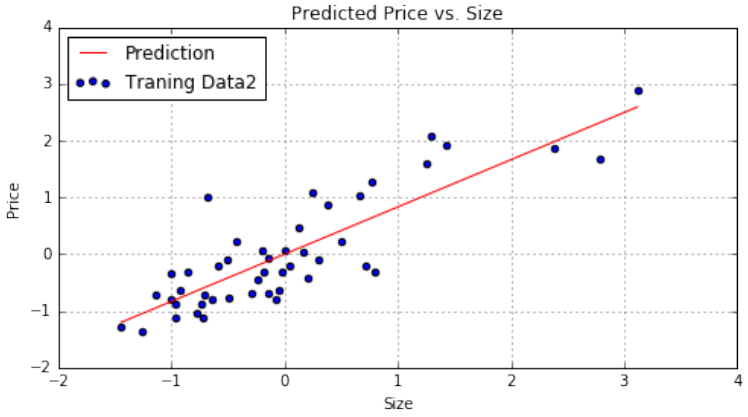


Figura 4: Linearização dos pontos

Podemos observa que o resultado foi o esperado, uma reta que aproxima bem os valores do preços da casa, há alguns pontos bem longe da reta, estes apresentam um resíduo maior em relação aos demais pontos.

Analisando os resultados obtidos ao simular a função gradiente para mais de uma variável, figura 5, neste caso, o preço da casa está em função da quantidade de banheiros.

Podemos observa que os pontos estão na vertical, com isso, obtivemos uma reta que não é satisfatória para essa variável, portanto, a relação de preço e banheiros não é o suficiente para determinar o preço da casa.

### 4.4 Função Custo

Analisando os resultados obtidos ao simular a função gradiente para várias variáveis, obtivemos um array com os custos a da iteração, figura 6.

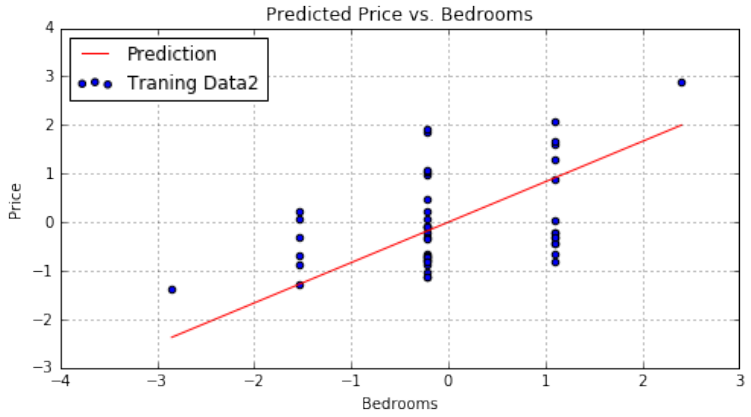


Figura 5: Linearização dos pontos

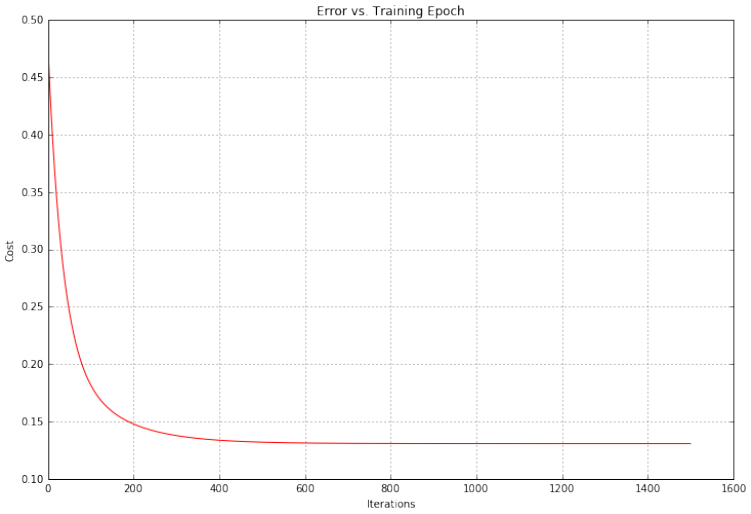


Figura 6: Linearização dos pontos

Percebemos que os custos a cada iteração diminui como o esperado, portanto podemos concluir que a relação preço e tamanho da casa, é uma boa aproximação, com isso, a curva continua tendo a forma de uma exponencial decrescente.

### 4.5 Um modelo do mundo real

Neste modelo não obtive um resultado satisfatório, esse resultado pode ser explicado pela relação entre variável independente e variável dependente que não foram bem relacionadas. Observando a curva apresenta na linearização figura 7, para os pontos na direção da linearização, mostram-se satisfatórios, porém, no contexto geral o resultado não é bom.

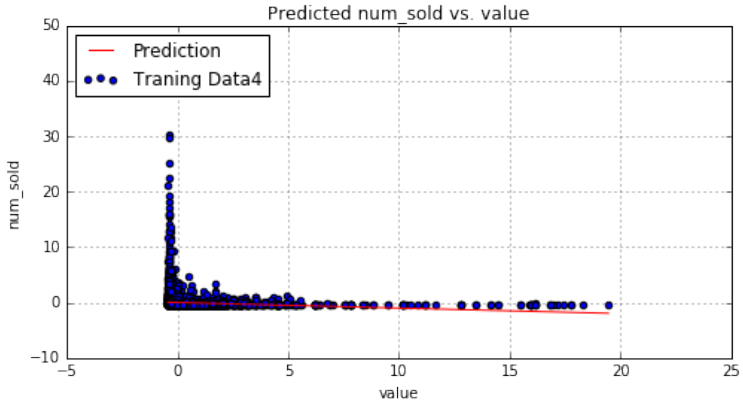


Figura 7: Linearização dos pontos

# Apêndices

## A Implementação da função de custo

```
def compute_cost(X, y, beta):
    m = len(X)
    hb = np.power((X.dot(beta.T) - y),2)
    return np.sum(hb) / (2 * m)
```

## B Implementação do Gradiente descendente

```
def gradient_descent(X, y, theta, alpha, iters):
    '''
    alpha: learning rate
    iters: number of iterations
    OUTPUT:
    theta: learned parameters
    cost:  a vector with the cost at each training iteration
    '''
    temp       = np.matrix(np.zeros(theta.shape))
    parameters = int(theta.ravel().shape[1])
    cost       = np.zeros(iters)
    n = len(X)
    for i in range(iters):
        for j in range(parameters):
            gradient = np.multiply((X.dot(theta.T) - y),X[:,j])
            temp[0,j] -= (alpha/(n)) * np.sum(gradient)

        theta = temp #update
        cost[i] = compute_cost(X, y, theta)
    return theta, cost
```

## Referências


[1] Coursera - curso de machine learnig. https://pt.coursera.org/learn/machine-learning/lecture/kCvQc/gradient-descent-for-linear-regression. Acesso: 2017-04-01.

[2] Gradient descent - wikipedia. https://en.wikipedia.org/wiki/Gradient_descent. Acesso: 2017-04-04.

[3] Gradient descent - wikipedia. http://www.onmyphd.com/?p=gradient.descent. Acesso: 2017-04-03.

[4] Linear regression. http://onlinestatbook.com/2/regression/intro.html. Acesso: 2017-04-02.

[5] Yale - linear regression. http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm. Acesso: 2017-04-02.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[7] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., 2001.


2 1 3 5 4 6 7