

## MACHINE LEARNING FOR BIOMEDICAL DATA

# Linear Regression with one or more variables

How to use Linear Regression for outcome prediction in Medicine



Luca Zammataro

Dec 9, 2019 · 14 min read ★

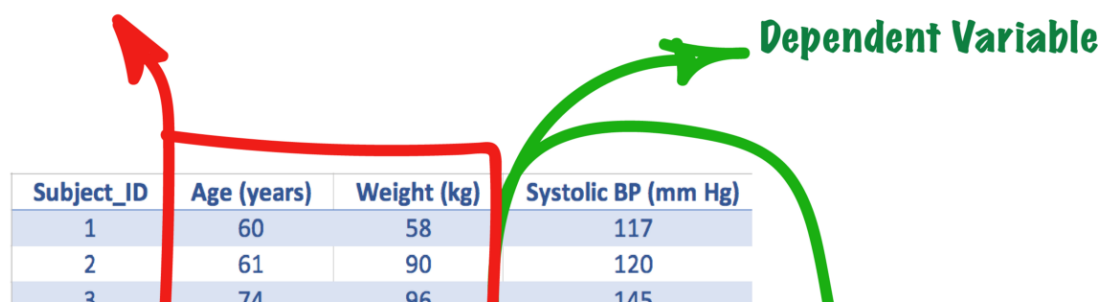
## Introduction

Linear Regression is a statistical model used in Machine Learning that falls in the “Supervised Learning” class of algorithms, and it applies to the analysis of biomedical data. We use it for predicting continuous-valued outputs, differently from the Logistic Regression, that instead is used for predicting discrete-valued outputs (i.e., classification).

Before starting, I suggest readers following the interesting course in Machine Learning at Coursera by Andrew NG [1]. The course provides an excellent explanation of all the arguments treated in this post.

Let's start with our example: we have a dataset of patients checked for Systolic Blood Pressure (SBP) and monitored for age and weight, and we want to predict the SBP for a new patient. Implicitly we hypothesize that factors such as weight and age influence the SBP. For our dataset, we will take values of a Table from [2]

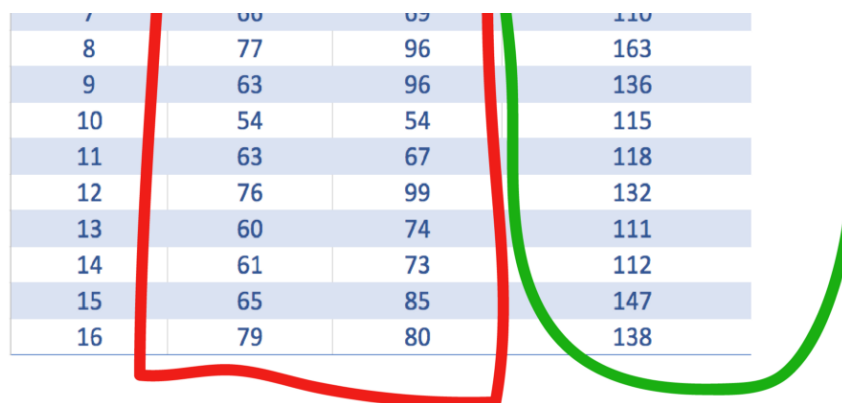
### Independent Variables



Subject_ID	Age (years)	Weight (kg)	Systolic BP (mm Hg)
1	60	58	117
2	61	90	120
3	74	96	145

You have two free stories left this month.  
[Upgrade for unlimited access](#)





7	66	69	110
8	77	96	163
9	63	96	136
10	54	54	115
11	63	67	118
12	76	99	132
13	60	74	111
14	61	73	112
15	65	85	147
16	79	80	138

Table 1: Dataset

The Variable to be explained (SBP) is called the **Dependent Variable**, or **Response Variable**, and it matches with our **output** variable or **target vector**. Instead, the variables that explain the **input** (age and weight) are called **Independent Variables** or **Predictor Variables**, or **Features**. If the dependent and independent variables are continuous, as is the case for *SBP*, *age*, and *weight*, then a **Correlation coefficient** can be calculated as a measure of the strength of the relationship between them. [3]

We say that **Linear Regression** represents an evolution of the **Correlation**. The difference between them is: Correlation refers to the strength of the relationship between two or more variables. The Regression, instead, refers to an ensemble of statistical techniques and algorithms for describing the relationship between two or more variables [2].

Linear Regression assumes that the relationship between one or multiple input features and the relative target vector (outputs) is approximatively linear. [4], and it enables the identification and characterization of this relationship. The consequence of this assumption is that, in the Linear Regression model, the input features have an “effect” on the target vector (output), and this effect is constant.

The “effect” is often identified as “coefficient”, “weight” or “parameter”, and more simply, we say that Linear Regression computes a weighted sum of the input features, plus a constant called “bias term” or intercept [5].

To simplify, let’s start with the application of the Linear Regression with one variable.

## Linear Regression with One Variable

Starting One Variable is a fundamental step if we want to understand thoroughly how

You have two free stories left this month.

[Upgrade for unlimited access](#)



All the code presented in this post is written in Python 2.7, and it's self-explicative also for the porting to many other languages. For the implementation environment, I suggest the use of [Jupyter Notebook](#).

Linear Algebra calculation will be primarily used, because of one the intrinsic advantages in avoiding, where possible, 'while' and 'for' loops. To accomplish this goal, we will use [NumPy](#), a robust library of math functions for scientific computing with Python.

Before starting with the description of the Linear Regression model, it's necessary to take a glance at our data and try to understand whether Linear Regression could apply to the data. The aim is predicting, for example, systolic pressure value, based on the patient age.

## Step 1: Importing Python libraries

First of all, we import all the packages required for the Python code that will be discussed in this post: **NumPy**, **Pandas** and **matplotlib**. These packages belong to [SciPy.org](#), which is a Python-based ecosystem of open-source software for mathematics, science, and engineering.

Numpy is necessary for the Linear Algebra calculations.

Pandas is an open-source library providing high-performance, data structures, and data analysis tools for Python. A pandas DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure. It consists of three principal components, the data, rows, and columns, with labeled axes (rows and columns). Follow this [link to the GeeksForGeeks portal](#), for having access to detailed information about the use of the pandas DataFrame.

Matplotlib is fundamental for creating all the plots.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4 from __future__ import division
```

ImportPackages\_1.py hosted with ❤ by GitHub

[view raw](#)

You have two free stories left this month.  
[Upgrade for unlimited access](#)



## Step 2: Creating the dataset

In this step, we will create a dataset with our values. An example of the comma-separated-values format, with a header, is the following:

**Age,Weight,SBP**

60,58,117

61,90,120

74,96,145

57,72,129

63,62,132

68,79,130

66,69,110

77,96,163

63,96,136

54,54,115

63,67,118

76,99,132

60,74,111

61,73,112

65,85,147

79,80,138

Copy and paste the values in a file, and save it as “SBP.csv”

## Step 3: Opening the dataset

Once we have created the SBP.csv dataset, upload it and create a DataFrame using the Pandas *pd* object. The Python code for the dataset uploading is the following:

```
1 # Read data from file 'SBP.csv'
2 # Make a pandas DataFrame "df"
3 # containing the SBP data.
4
5 df = pd.read_csv("SBP.csv")
```

CreatePandasDataFrameFromCSV\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 2: upload data and create a pandas DataFrame

You have two free stories left this month.  
[Upgrade for unlimited access](#)

×

	<b>Age</b>	<b>Weight</b>	<b>SBP</b>
<b>0</b>	60	58	117
<b>1</b>	61	90	120
<b>2</b>	74	96	145
<b>3</b>	57	72	129
<b>4</b>	63	62	132
<b>5</b>	68	79	130
<b>6</b>	66	69	110
<b>7</b>	77	96	163
<b>8</b>	63	96	136
<b>9</b>	54	54	115
<b>10</b>	63	67	118
<b>11</b>	76	99	132
<b>12</b>	60	74	111
<b>13</b>	61	73	112
<b>14</b>	65	85	147
<b>15</b>	79	80	138

Table 2: Visualizing a pandas DataFrame

You have two free stories left this month.  
[Upgrade for unlimited access](#)



The SBP dataset is formed by 3 columns (Age, Weight, and SBP), but we will upload the first and the last columns (Age and SBP); our model will determine the strength of the relationship between Age and SBP. Pandas makes easy accessing to DataFrame variables, that will be copied in an  $X$  vector, containing the input, and a  $y$  vector, for the output.

```
1 # Make the X numpy arrays, containing the values of Age.
2 # and the output vector y, containing the SBP values.
3
4 X = np.asarray(df.Age.values)
5 y = np.asarray(df.SBP.values)
```

UploadingDataset\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 3: Uploading the dataset

## Step 5: Feature Scaling and Normalization

Before going ahead, the SBP dataset reported here has to be rescaled and normalized. In Machine Learning, scaling and normalization are required, especially when discrepancies in order of magnitude among the features and the output occurs. As mentioned in the Introduction, we will use Linear Algebra on matrixes to avoid, where possible, *while* and *for* loops on variables. Moreover, programming Linear Algebra will result in good readability of the code.

For this purpose, In Python, we can take advantage of NumPy. As mentioned before, NumPy is a library of math functions for scientific calculation and for Linear Algebra. All the operations can be simplified, creating a NumPy object, and using some of the associated methods. The reason why we want to use NumPy is that, even though many operations on matrixes can be done using the regular operators (+, -, \*, /), NumPy guarantees a better control over the operations, especially if the matrixes are big. For example, with NumPy, we can multiply arguments element-wise, like the features matrix  $X$  and the output vector  $y$ :

```
1 np.multiply(X, y)
```

MultiplyArgumentsElement-wise.py hosted with ❤ by GitHub

[view raw](#)

Code 4: Multiply arguments element-wise

You have two free stories left this month.  
[Upgrade for unlimited access](#)



```

1  def FeatureScalingNormalization(X):
2      # Initialize the following variables:
3      # Make a copy of the X vector and call it X_norm
4      X_norm = X
5
6      # mu: It will contain the average
7      # value of X in training set.
8      mu = np.zeros(X.shape[0])
9
10     # sigma: It will contain the Range(max-min)
11     # of X or Standard Deviation
12     sigma = np.zeros(X.shape[0])
13
14     mu = X.mean()
15     # The Standard Deviation calculation with NumPy,
16     # requires the argument "degrees of freedom" = 1
17     sigma = X.std(ddof=1)
18
19     # number of training examples
20     m = X.shape[0]
21
22     # Make a vector of size m with the mu values
23     mu_matrix = np.multiply(np.ones(m), mu).T
24
25     # Make a vector of size m with the sigma values
26     sigma_matrix = np.multiply(np.ones(m), sigma).T
27
28     # Apply the Feature Scaling Normalization formula
29     X_norm = np.subtract(X, mu).T
30     X_norm = X_norm /sigma.T
31
32     return [X_norm, mu, sigma]

```

Code 5: The FeatureScalingNormalization() function.

Code 5 implements a Python function called *FeatureScalingNormalization()*. This function takes *X* that is the features vector as an argument, and return 3 arguments: 1) the same *X* vector but scaled and normalized (*X\_norm*), 2) *mu*, that is the average values of *X* in training set) and 3) *sigma* that is the Standard Deviation. Also, we will store *mu* and *sigma* because these parameters will be fundamental later. Copy the following code and paste it in a new Jupyter Notebook cell:

```
4
5 # get the normalized X matrix
6 X = np.asarray(featuresNormalizerresults[0]).T
7
8 # get the mean
9 mu = featuresNormalizerresults[1]
10
11 # get the sigma
12 sigma = featuresNormalizerresults[2]
```

RunFeatureScalingNormalizationOneVariable\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 6: Run the FeatureScalingNormalization function.

Typing 'X' in a Notebook cell will display the new values of X:

```
array([-0.73189052, -0.59728997,  1.15251726, -1.13569219,
       -0.32808885,  0.34491392,  0.07571281,  1.55631892, -0.32808885,
       -1.53949386, -0.32808885,  1.42171837, -0.73189052, -0.59728997,
       -0.05888774,  1.82552004])
```

The X vector containing the Age values is now normalized.

## Step 6: Add a column of ones to the X vector

Now we will add a column of ones to the X vector.

```
1 # Add a column of ones to the matrix X
2
3 m = len(y) # number of training examples
4 X = np.vstack((np.ones(m), X.T)).T
```

AddColumnOnes\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 7: Add a column of ones to the X vector.

This is the new structure of X:

```
array([[ 1.          , -0.73189052],
       [ 1.          , -0.59728997],
       [ 1.          ,  1.15251726],
       [ 1.          , -1.13569219],
       [ 1.          , -0.32808885],
       [ 1.          ,  0.34491392],
       [ 1.          ,  0.07571281],
       [ 1.          ,  1.55631892],
       [ 1.          , -0.32808885],
       [ 1.          , -1.53949386],
       [ 1.          , -0.32808885],
       [ 1.          ,  1.42171837],
       [ 1.          , -0.73189052],
       [ 1.          , -0.59728997],
       [ 1.          , -0.05888774],
       [ 1.          ,  1.82552004]])
```

You have two free stories left this month.

[Upgrade for unlimited access](#)





```
[ 1.      , -0.32808885],  
[ 1.      , -1.53949386],  
[ 1.      , -0.32808885],  
[ 1.      ,  1.42171837],  
[ 1.      , -0.73189052],  
[ 1.      , -0.59728997],  
[ 1.      , -0.05888774],  
[ 1.      ,  1.82552004]])
```

## Step 7: Plotting the dataset

Plotting our data is a useful practice when we want to have an idea of how they are distributed. Plot the data using the matplotlib scatter method:

```
1 # Plot the data (plt is a matplotlib object)  
2 plt.scatter(X[:,[1]], y, color='black')  
3  
4 # Put labels  
5 plt.xlabel("Age")  
6 plt.ylabel("SBP")
```

PlotData\_a\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 8: Plot data

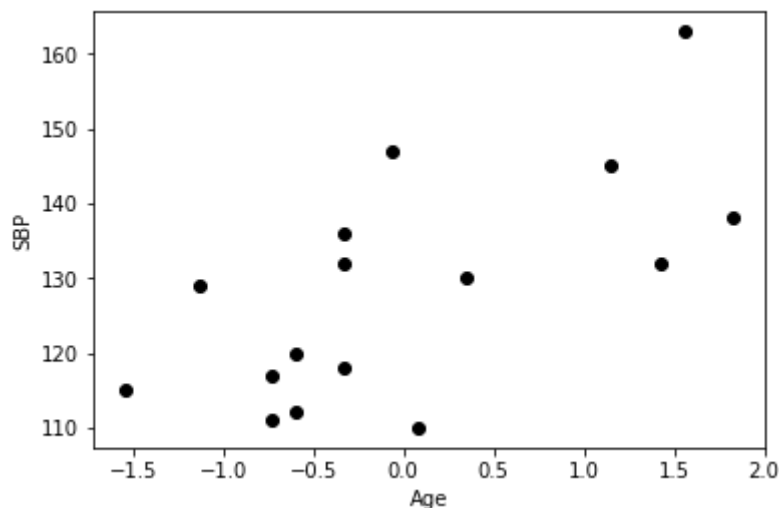


Figure 1: Plot Age-SBP

Visualizing the data at a glance, we can notice a pattern of increasing relationship, between *Age* and *SBP*. This is what we expect since systolic pressure is physiologically connected to age increasing.

You have two free stories left this month.  
[Upgrade for unlimited access](#)



The idea underlying the Linear Regression is represented by a function that predicts the output  $y$  based on the input feature  $X$ .

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

Equation 1: Linear Regression Model

The predicted output is the  $\mathbf{h} = \boldsymbol{\theta} * \mathbf{X}$  term that is equal to a constant called “bias term” or “intercept term” or  $\boldsymbol{\theta}_0$  plus a weighted sum of the input features  $X$ , where  $\boldsymbol{\theta}_1$  represents the weight for  $X$ . We will call this function “Hypothesis”, and we will use it to “map” from  $X$  (Age) to  $y$  (SBP).

Since we are using Linear Algebra, for all the calculation, we can write the Hypothesis model in the vectorized form:

$$h_{\theta}(x) = [\boldsymbol{\theta} \mathbf{X}]$$

Equation 2: Linear Regression Model in vectorized form

where  $\boldsymbol{\theta}_0$  and  $\boldsymbol{\theta}_1$  are expressed as vector  $\boldsymbol{\theta} = [\boldsymbol{\theta}_0, \boldsymbol{\theta}_1]$ , and the Hypothesis is equal to  $\boldsymbol{\theta} \mathbf{X}$ .

The best performance in predicting  $y$  consists of finding  $\theta$  values for which the distance between the predicted  $y$  value and the actual  $y$  value is closer to the minimum.

```

1  # Plot the Hypothesis: set theta manually
2  theta_0 = 140.0
3  theta_1 = 5.0
4
5  # the vector theta is initialized with theta_0 and theta_1
6  theta = np.asarray([theta_0,theta_1]).astype(float)
7
8  # Plot the data
9  plt.scatter(X[:,[1]], y, color='black')
10
11 # Plot a red line corresponding to our Hypothesis model.
12 plt.plot(X[:,[1]], np.sum(np.multiply(X,theta), axis=1), color='red', linewidth=1)
13
14 # Put labels
15 plt.xlabel("Age")
16 plt.ylabel("SBP")

```

PlotHypothesisWithThetaManually\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 9: Plot the Hypothesis with  $\theta = [140.0, 5.0]$

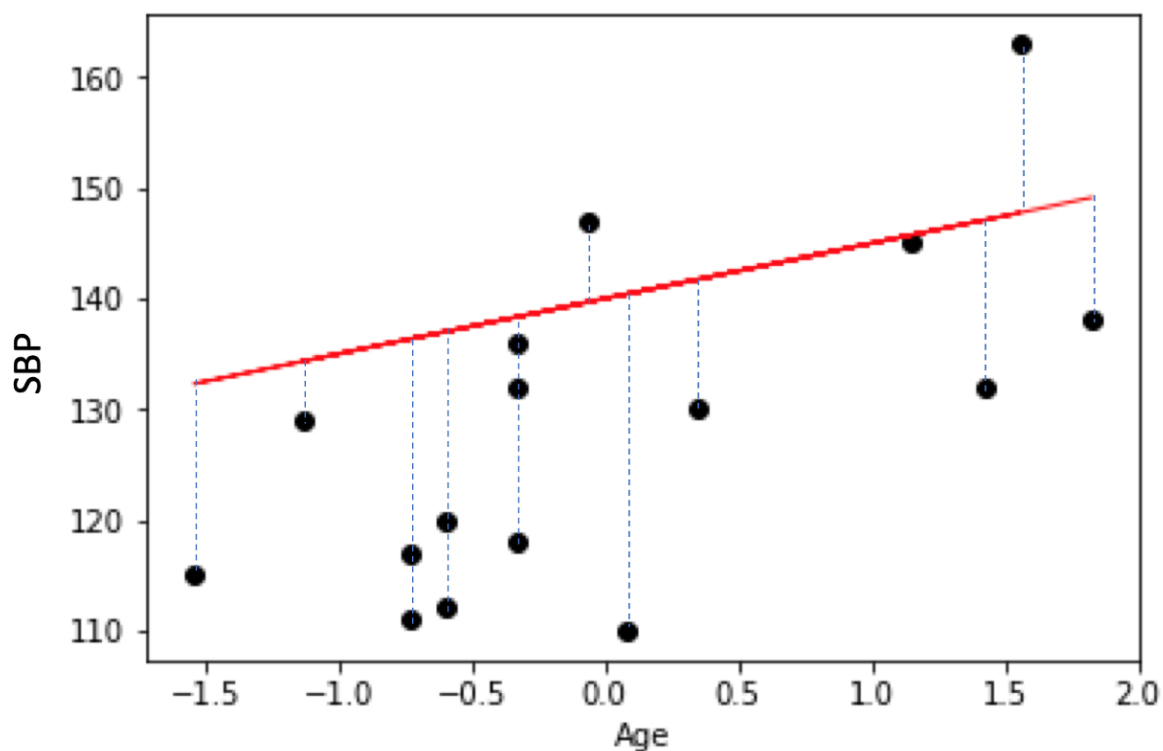


Figure 2:  $\theta = [140.0; 5.0]$

But this model, obviously, does not fit our data. As highlighted by the blue lines connecting dots with the red line, the Hypothesis “touches” some of the  $y$  values, but the rest of the  $h$  vector is far from the minimum. So we are tempted to guess which  $\theta$  could predict  $y$  when setting with different values. We could choose  $\theta$  “by trial and error” to minimize all the distances between the *Hypothesis* and  $y$ . To accomplish this goal, we can calculate the **Cost Function** for our model.

## Step 9: Calculating the Cost Function

The **Cost Function** can register how much far we are from the minimum of the Hypothesis model and can help us in finding the best  $\theta$ . The equation describing the Cost Function is the following:

$$J = \frac{1}{2m} \sum_{i=1}^m \overbrace{\left( h_{\theta}(x^{(i)}) - y^{(i)} \right)^2}^{\text{Squared Error}}$$

Equation 3: Linear Regression Cost Function

Where  $m$  is the length of the  $X$  vector (in our case = 16), and  $i$  is the index assigned to each item in the dataset. The Equation is composed of three components:

1. the Hypothesis ( $h = \theta X$ )
2. the SquaredError that is  $= (h - y)^2$
3. The Cost Function  $J$  that is then calculated as  $J = 1/2m * \text{Sum}(\text{SquaredError})$

Since we use Linear Algebra, the vectorized implementation of *Equation 3* is the following:



## Equation 4: Linear Regression Cost Function (vectorized form)

**Intuition I.**

In order to simplify the explanation, let's try to manually calculate the *Cost Function* for a smaller dataset composed only of the first 3 values of the SBP dataset, and with  $\theta = [120.0, 10.0]$ . These parameters are chosen randomly, at the moment, because we don't have to set the best  $\theta$  for now. We will split the  $X$  and  $y$ , producing arrays  $X_1$  and  $y_1$ :

```
1 # Make a vector X_1 with the first 3 values of X
2 X_1 = X[0:3]
3
4 # Make a vector y_1 with the first 3 values of y
5 y_1 = y[0:3]
6
7 # Set m_1 = 3
8 # number of training examples
9 m_1 = 3
```

MakeVectorsX1andy1\_1.py hosted with ❤ by GitHub

[view raw](#)Code 10: Make a vector  $X_1$  and  $y_1$  with the first 3 values of  $X$ , and  $y$ .

Also, we have to set  $m=3$ , because we have three samples now. Let's plot the data and the Hypothesis as following:

```
1 # Plot data and Hypothesis: set theta manually
2
3 theta_0 = 120.0
4 theta_1 = 10.0
5
6 # the vector theta is initialized with theta_0 and theta_1
7 theta = np.asarray([theta_0, theta_1]).astype(float)
8
9 # Plot the data
10 plt.scatter(X_1[:, [1]], y_1, color='blue')
11
12 # Plot a red line corresponding to our Hypothesis model.
```

You have two free stories left this month.  
[Upgrade for unlimited access](#)



```

16
17 # Put labels
18 plt.xlabel("Age")
19 plt.ylabel("SBP")

```

PlotDataHypothesisWithThetaManually 1.py hosted with ❤ by GitHub

[view raw](#)

Code 11: Plot data and Hypothesis

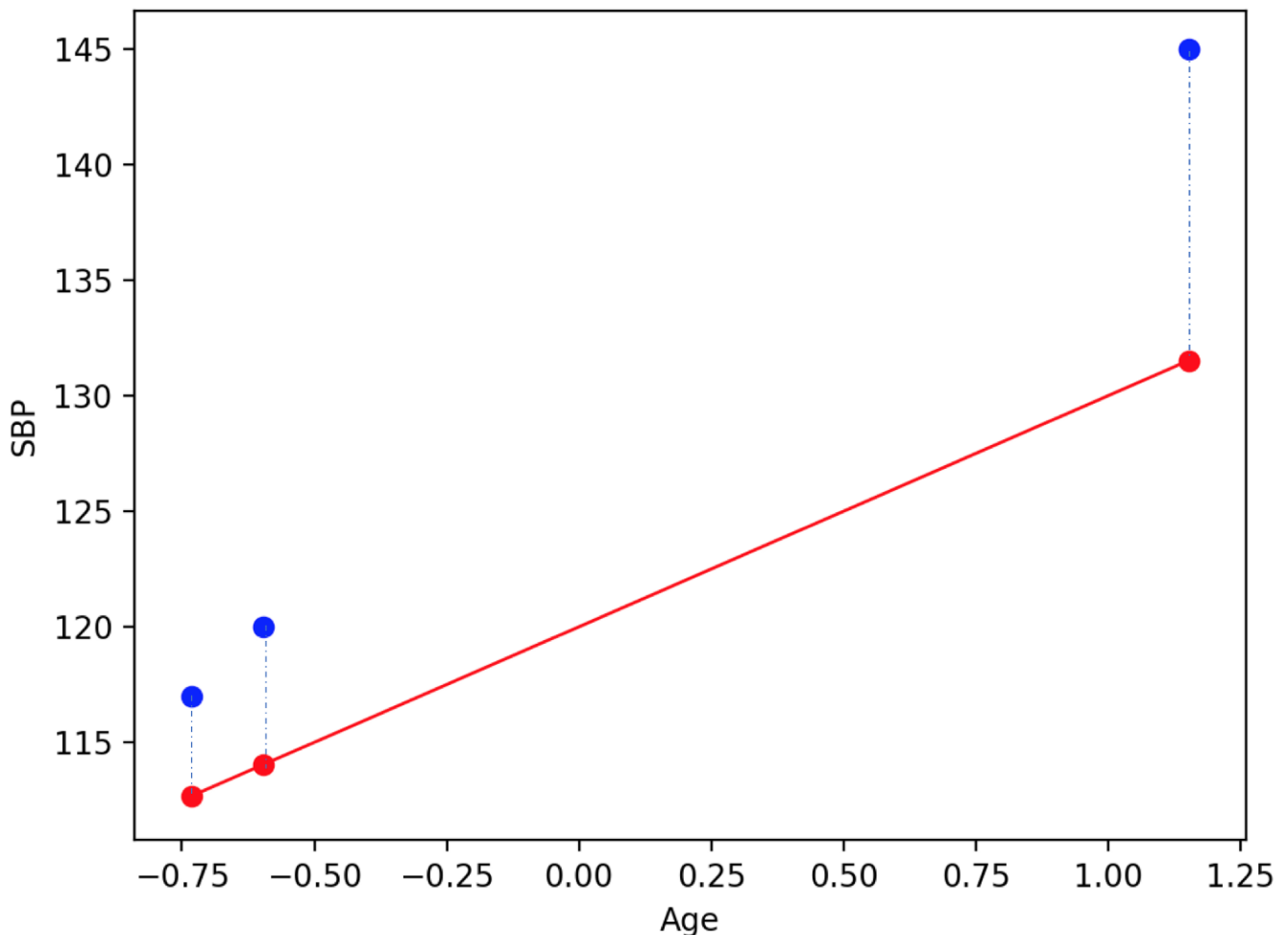


Figure 4: Plot of the first 3 values of the dataset;  $\theta = [120.0; 10.0]$

The vector  $y$  corresponding to the first three values (the blue dots) of the SBP is:

$$y = [117.0, 120.0, 145.0]$$

Since our  $\theta$  is  $= [120, 10.0]$ , the product of  $h = \theta * X_1$  will be represented by the following vector, (highlighted by the dots on the red line):

$$h = \theta * X_1 = [112.7, 114.0, 131.5]$$

The dashed blue lines highlight the distances between actual  $y_1$  values and predicted

You have two free stories left this month.  
[Upgrade for unlimited access](#)

×

$$\begin{aligned}
 J &= \frac{1}{2m} \sum_{i=1}^m \overbrace{(h_{\theta}(x^{(i)}) - y^{(i)})^2}^{\text{Squared Error}} = \\
 &= \frac{1}{2m} * [(112.7 - 117)^2 + (114.0 - 120)^2 + (131.5 - 145)^2] = \\
 &= \frac{1}{2m} * [(-4.3)^2 + (-6.0)^2 + (-13.5)^2] = \\
 &= \frac{1}{2m} * [18.65 + 35.7 + 181.6] = \\
 &= \frac{1}{2m} * 235.9 = \\
 &= \frac{235.9}{2*3} = 39.3
 \end{aligned}$$

Solution I: Calculating the Cost Function

...the Cost Function (J) is = 39.3

## The Cost Function in Python.

The following Python code implements the *Cost Function*:

```

1  # Calculate the Cost Function
2  def calcCostFunction(X, y, theta):
3
4      # number of training examples
5      m = len(y)
6
7      # initialize J (the cost)
8      J = 0
9
10     # Calculate h = X * theta (we're using vectorized Equation 4)
11     h = np.sum(np.multiply(X, theta), axis=1)
12
13     # Calculate the Squared Error = (h - y)^2 (vectorized)
14     SquaredError = np.power(np.subtract(h,y), 2)
15

```

You have two free stories left this month.  
[Upgrade for unlimited access](#)



```
19     return J
```

CalcCostFunction 1.py hosted with ❤ by GitHub

[view raw](#)

## Code 12: The code for calculating the Cost Function

The code implements step by step the Cost Function described in *Equation 4* (*vectorized*). Let's repeat again:

1. the Hypothesis ( $h = \theta X$ )
2. the SquaredError that is  $= (h - y)^2$
3. the Cost Function  $J$  that is  $= 1/2m * \text{Sum}(\text{SquaredError})$

Now that we have understood the mechanism underlying the *Cost Function* calculation, let's go back to the complete SBP dataset (16 patients). If we want to calculate the Cost Function for the whole SBP dataset, using  $\theta = [140.0; 5.0]$ , we will type :

```
1  calcCostFunction(X,y,theta)
```

RunCalcCostFunction\_1.py hosted with ❤ by GitHub

[view raw](#)

## Code 13: Running calcCostFunction

The function will return  $J = 138.04$ , which is the *Cost Function* calculated for  $\theta = [140.0; 5.0]$ . This  $J$  is not the minimum  $J$  that we could find, since that we have manually set  $\theta$ , without any idea about how to minimize it. The following Intuition II could help us in understanding better the limit of our manual approach.

## Intuition II.

The following code generates randomly 10  $\theta$  vectors and passes them to the *calcCostFunction*, producing a table of the relative Cost Functions ( $J$ ):

```
1  # Try some random theta, and produce a table with
2  # random theta values and their relative J
3
4  import random # import the random library
5
6  print "[Th0 Th1]", "\tJ" # write an header
7
```

You have two free stories left this month.  
[Upgrade for unlimited access](#)





```

11 theta = np.asarray([theta_0, theta_1]).astype(float)
12 # Calculate J and print the table
13 print(theta, calcCostFunction(X, y, theta))

```

TryRandomThetaAndCalculateCostFunction\_1.py hosted with ❤ by GitHub

[view raw](#)Code 14: Try random  $\theta$  and calculate the Cost Function

The resulting output is:

```

[Th0 Th1] J
[38. 55.] 5100.4688623710845
[71. 47.] 2352.6631642080174
[28. 76.] 7148.466632549135
[73. 75.] 3579.826857778751
[79. 47.] 1925.1631642080174
[12. 42.] 7320.026790356101
[68. 25.] 1992.2131192595837
[25. 92.] 8565.015528875269
[51. 46.] 3667.1483894376343
[13. 62.] 7992.509785763768

```

The “take-home message” is that trying to handly minimize  $J$ , is not the correct way to proceed. After 10 runs on randomly selected  $\theta$ 's, the behavior of  $J$  is unpredictable. Moreover, there is no way to guess  $J$  basing on  $\theta$ . So the question is: How we can choose  $\theta$ , to find the minimum  $J$ ? We need an algorithm that can minimize  $J$  for us, and this algorithm is the argument of the next Step.

## Step 10: Gradient Descent

We are interested in finding the minimum of the *Cost Function* using **Gradient Descent**, which is an algorithm that can automatize this search. The Gradient Descent calculates the derivative of the *Cost Function*, updating the vector  $\theta$  by mean of the parameter  $\alpha$ , that is the learning rate. From this moment on, we will refer to the SBP dataset, as the **training set**. This clarification is essential since *Gradient Descent* will use the difference between the actual vector  $y$  of the dataset and the  $h$  vector prediction, to “learn” how to find the minimum  $J$ . The algorithm will repeat until it will converge.  $\theta$  updating has to be simultaneous.

**Repeat until converge {**

1  $m$  **Error**

You have two free stories left this month.

[Upgrade for unlimited access](#)

✕

$$\theta_1^{new} = \theta_1^{old} - \alpha \frac{1}{m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)}) x^{(i)}$$

*} update  $\theta_0$  and  $\theta_1$  simultaneously*

Equation 4: Gradient Descent implementation

Since we use Linear Algebra, the vectorized implementation is the following:

$$\theta^{new} = \theta^{old} - \alpha \frac{1}{m} \sum_{i=1}^m [(\overbrace{h_{\theta}X - y}^{\text{Error}})X^T]$$

Equation 5: Gradient Descent (vectorized form)

Note that here we have to transpose  $X$  since  $X$  is a  $[16, 2]$  matrix and *Error* is a  $[16, 1]$  vector.

### Gradient Descent implementation.

The following Python code implements the Gradient Descent. We will use the vectorized form of *Equation 5*:

```
1 def gradientDescent(X, y, theta, alpha, num_iters):
2
3     # number of training examples
4     m = len(y)
5
6     # Initialize J_history and Theta_history
7     J_history = []
8     Theta_history = []
9
10    for i in range(num_iters):
11
12        # Calculate h = X * theta (vectorized Eq. 5)
13        h = np.sum(np.multiply(X, theta), axis=1)
```

You have two free stories left this month.  
[Upgrade for unlimited access](#)



```

17
18     # Calculate the new theta (vectorized Eq. 5)
19     theta_new = alpha * 1/m * np.sum(np.multiply(X.T, error), axis=1)
20
21     # Update theta
22     theta = np.subtract(theta, theta_new)
23
24     # Collect all the theta and J
25     Theta_history.append(theta.tolist())
26     J_history.append(calcCostFunction(X,y,theta).tolist())
27
28
29     return theta, Theta_history, J_history

```

CODE 13: THE GRADIENT DESCENT FUNCTION

To run the *Gradient Descent*, we have to initialize  $\theta$ , *iterations*, and  $\alpha$ , that together with  $X$  and  $y$  are the arguments of the *gradientDescent* function:

```

1  # Running the Gradient Descent
2
3  # Initialize theta
4  theta = np.asarray([0,0]).astype(float)
5
6  # Set the number of iterations for the Gradient Descent
7  iterations = 2000
8
9  # Set the Learning Rate
10 alpha = 0.01
11
12 # Run the gradientDescent() function, and collect the output in "results"
13 results = gradientDescent(X, y, theta, alpha, iterations)
14
15 # Get the theta from the results
16 theta = results[0] # new theta
17
18 # Get the theta history
19 Theta_history = results[1] # Theta history
20
21 # Get the J history
22 J_history = results[2] # Cost funtion history

```

RunningGradientDescent 1.py hosted with ❤ by GitHub

[view raw](#)

You have two free stories left this month.  
[Upgrade for unlimited access](#)



The results are collected in the “results” list. This list is composed of the found  $\theta$ , plus two lists containing the  $\theta$  and  $J$  histories. After 2000 iterations the *Gradient Descent* has found  $\theta = [128.4, 9.9]$ , and  $J = 59.7$ , which is the minimum  $J$ . We will use the two lists for plotting the *Gradient Descent* activity. The following code will plot the training set and  $h$ .

```

1 # Plot training set
2 plt.scatter(X[:,[1]], y, color='black')
3
4 # Plot Hypothesis (theta were calculated with the Gradient Descent)
5 plt.plot(X[:,[1]], np.sum(np.multiply(X,theta), axis=1), color='red', linewidth=1)
6
7 # Put labels
8 plt.xlabel("Age")
9 plt.ylabel("SBP")

```

PlotDatasetAndH\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 17: Plot dataset and  $h$  for  $\theta = [128.4, 9.9]$ .  $J = 59.7$  is the minimum

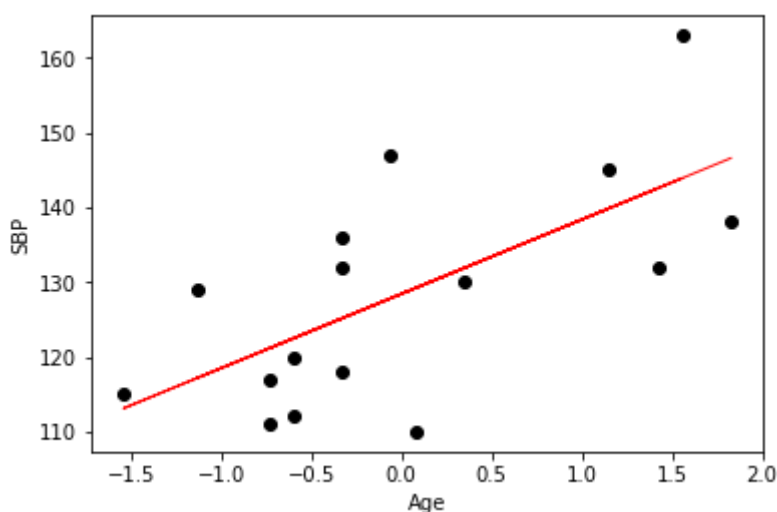


Figure 5: Plot of dataset and  $h$ ;  $\theta = [128.4; 9.9]$ ;  $J = 59.7$

The *Hypothesis*  $h$  now fits with our data!

Let's plot the  $\theta$  history:

```

1 # Plot the Theta history
2
3 theta_0 = np.asarray(Theta_history)[:,[0]]

```

You have two free stories left this month.  
[Upgrade for unlimited access](#)

×

```

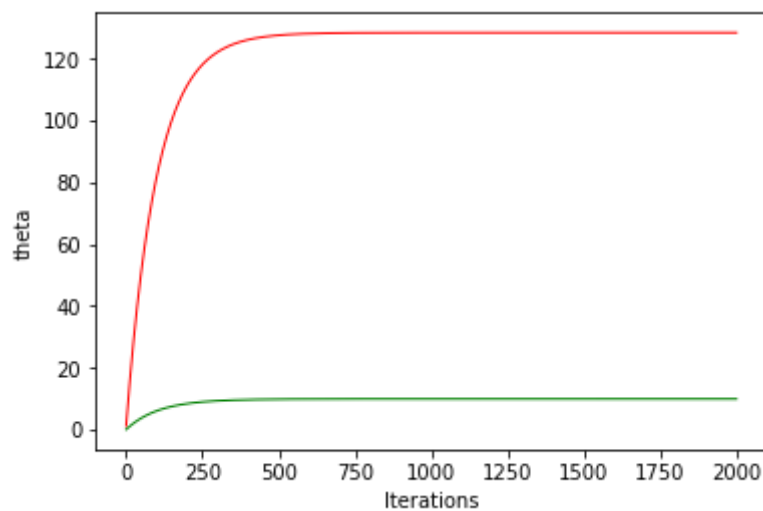
1  plt.plot(theta_1[0:len(theta_1)], color='green', linewidth=1)
8
9  # Put labels
10 plt.xlabel("Iterations")
11 plt.ylabel("theta")

```

PlotThetaHistory\_1.py hosted with ❤ by GitHub

[view raw](#)Code 18: Plot the  $\theta$  history

A plot of the  $\theta$  history is shown in *Figure 6*. The red curve represents  $\theta_0$ , the green curve  $\theta_1$ . After 2000 iterations,  $\theta$  is  $= [128.4; 9.9]$

Figure 6: The  $\theta$  history plot. The red curve represents  $\theta_0$ ; the green curve represents  $\theta_1$ 

Now let's plot the  $J$  history:

```

1  # Plot the J history
2  plt.plot(J_history[0:len(J_history)], color='blue', linewidth=1)
3
4  # Put labels
5  plt.xlabel("Iterations")
6  plt.ylabel("J")

```

PlotJHistory\_1.py hosted with ❤ by GitHub

[view raw](#)Code 19: Plot the  $J$  history

You have two free stories left this month.  
[Upgrade for unlimited access](#)



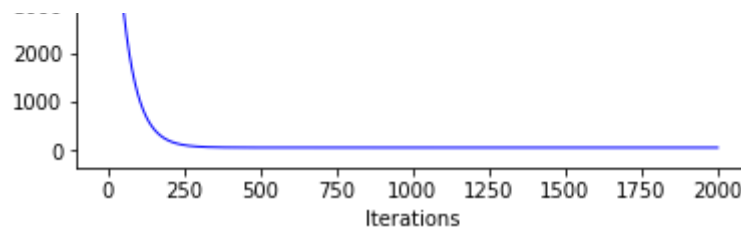


Figure 7: The J history plot

After circa 200 iterations the Cost Function falls down, stabilizing around 59.7 after 1500 iterations. The  $J$  curve depends on  $\alpha$ , that we have set to 0.01.

## Step 11: Predictions

Now that we have found the best  $\theta$ , we can predict the Systolic Blood Pressure for a 75 years-old person. The query is a vector, composed of two numbers  $[1, 75]$ . The first number corresponds to the feature  $x_0$ . To run the prediction, we have to scale and normalize the query vector, using the  $\mu$  and  $\sigma$  parameters that we have calculated in *Step 5: Feature Scaling and Normalization*. The query vector will be  $[1, 1.29]$ . Then, we have to multiply the *query* for the  $\theta$  vector ( $\theta = [128.4, 9.95]$ ). The following code implements the prediction.

```
1  # Perform prediction basing on a query
2
3  # Predict the Systolic Blood Pressure (SBP) for a 75 yo human:
4  query = [1, 75]
5
6  # Scale and Normalize the query
7  query_Normalized = [1, ((query[1]-mu)/sigma)]
8
9  # Predict the SBP: (the prediction is the product of "75" and theta)
10 prediction = np.sum(np.multiply(query_Normalized, theta))
```

PredictSBP\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 20: Predicting SBP

The SBP for a 75 years-old man is: 141.2

## Step 12: Intuitions concerning the learning constant $\alpha$

Let's do some experiments with the learning rate  $\alpha$ . Changing  $\alpha$  will affect the dynamics of  $J$ . If  $\alpha$  is too little, the *Gradient Descent* will converge slowly, and we need

You have two free stories left this month.  
[Upgrade for unlimited access](#)



the *Gradient Descent* converges, but for the first 40 iterations, the behavior of  $\theta$  is turbulent, for successively reaching stability. (Figure 8)

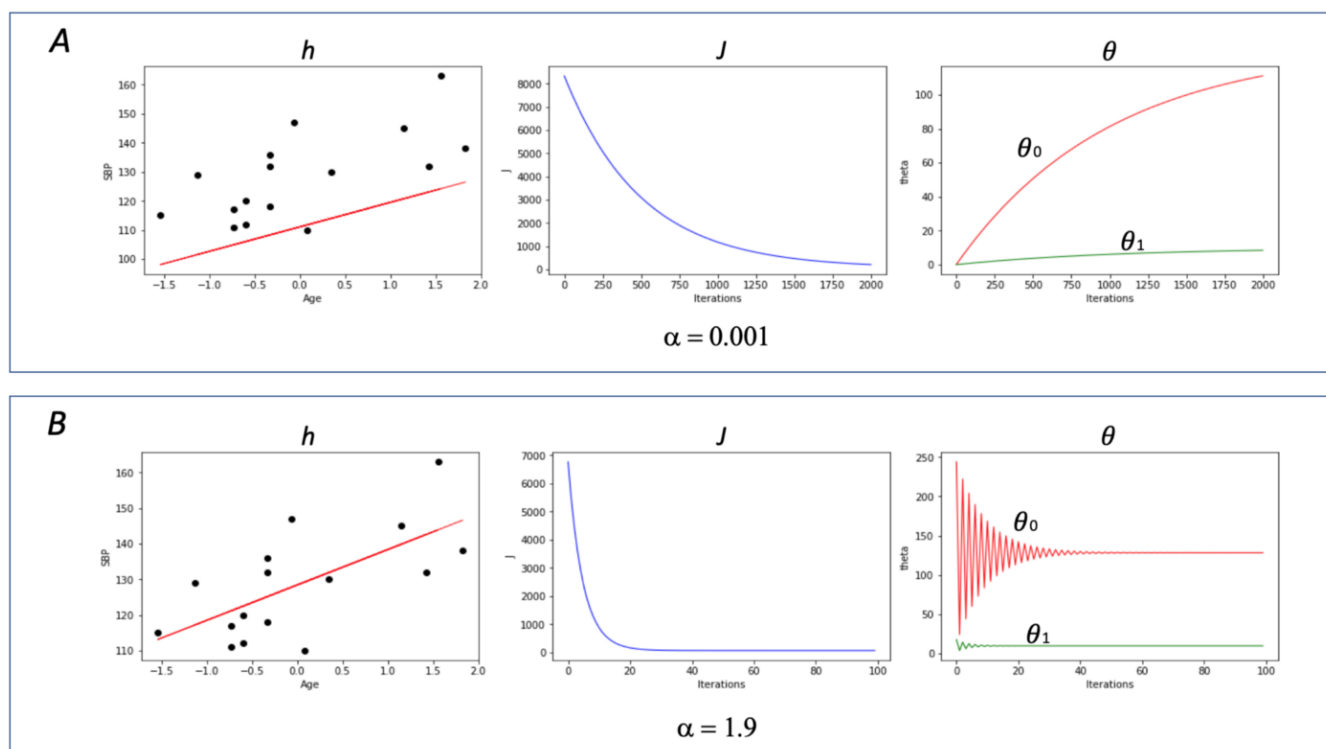


Figure 8: Experiments with  $\alpha$ . Panel A shows  $h$ ,  $J$  and  $\theta$ , with  $\alpha = 0.001$ . If  $\alpha$  is too little, 2000 iterations are not sufficient; Gradient Descent will be slow in converging, and it will require  $\sim 10000$  iterations for finding the minimum  $J$  (data not shown). Panel B shows the opposite situation. With  $\alpha = 1.9$ , Gradient Descent converges, but initially, the searching of  $\theta$  shows turbulence. After 40 iterations  $\theta$  reaches stability, and finally, the algorithm converges.

## Step 13: Contour plot of $J$ and $\theta$

We can create a contour plot, which is a graph containing many concentric tracks. For each track there are various pairs of  $\theta$  associated with a constant value of  $J$ . The  $\theta$  corresponding to the minimum  $J$ , lies at the center (the red dot). The other concentric lines correspond to all the different values of  $J$ . The most the distance from the center, the higher the value of the Cost Function  $J$ .

```
1 # Drawing a contour plot with all the theta.
2
3 # Create a space with all the theta
4 nslice = 50
5 theta0_vals = np.linspace(-100, 200, num=nslice)
6 theta1_vals = np.linspace(-400, 400, num=nslice)
7
```

You have two free stories left this month.  
[Upgrade for unlimited access](#)



```

11 for i in range(len(theta0_vals)):
12     for j in range(len(theta1_vals)):
13         t = np.asarray([theta0_vals[i], theta1_vals[j]]).astype(float)
14         J_vals.append(calcCostFunction(X, y, t).tolist())
15 J_vals = np.asarray(J_vals)
16 J_vals = J_vals.reshape((nslice, nslice)).T
17
18
19 levels = nslice
20 s = 1
21
22 # plot the contour with theta and J values
23 plt.contour(theta0_vals, theta1_vals, J_vals, levels)
24
25 # Draw the path of the Gradient Descent convergence
26 for k in range(0, iterations, 10):
27     plt.scatter(np.asarray(Theta_history)[k][0],\
28                np.asarray(Theta_history)[k][1], color='blue', s=s)
29
30 # Draw a red dot i correspondence of the theta associated to the minimum J
31 plt.scatter(theta[0], theta[1], color='red', s=10)
32
33 # Put labels
34 plt.xlabel("theta_0")
35 plt.ylabel("theta_1")

```

Code 20 produces the following plot:

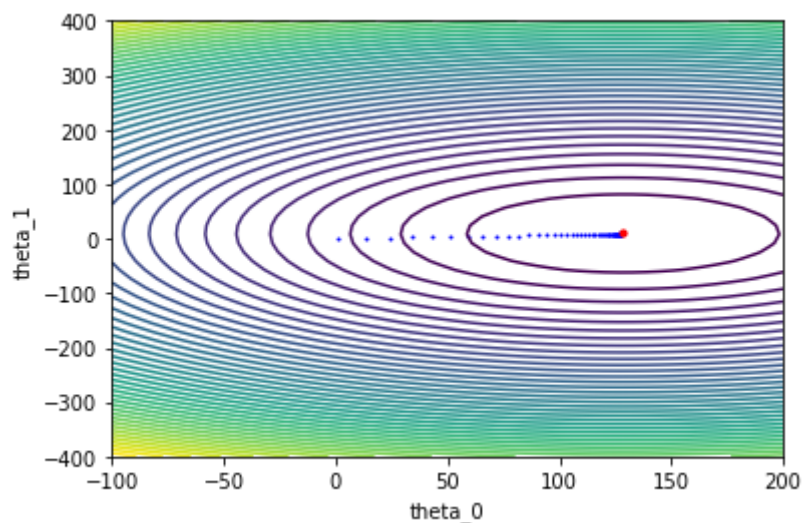


Figure 9: The contour plot of J and theta

You have two free stories left this month.  
[Upgrade for unlimited access](#)





Note that  $\theta = [128.4; 9.9]$  corresponding to the minimum  $J$  (59.7), is the red dot in the center of the graph. The blue dots track the path of the *Gradient Descent* in converging to the minimum.

## Step 14: How to adapt the code for Multiple Variables

We have explained the statistical mechanisms underlying *Linear Regression with one variable*: the feature *Age* in the SBP dataset. The major part of the code here proposed works also with multiple variables. The SBP dataset is composed of 2 features (*Age* and *Weight*) and one output: *SBP*. In this step, we will update the code in a way that it can fit with multiple variables. The only adjustments required concern:

1. The *Dataset uploading*
2. The *Feature Scaling and Normalization function*
3. The code for adding a column of “ones” to the vector  $X$
4. The *Prediction Query*.

### *Dataset uploading*

The code for uploading the dataset has to be modified in order to produce a new  $X$  vector containing *Age* and *Weight* of each patient:

```
1 # Make the X numpy arrays,
2 # containing the values of
3 # Age and Weight
4 # and the output vector y, containing SBP values.
5 X = np.vstack((np.asarray(df.Age.values), \
6               np.asarray(df.Weight.values)))
7 y = np.asarray(df.SBP.values)
```

UploadingDatasetMultiple\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 22: The code for uploading the dataset for multiple variables

The numpy method used for producing the new  $X$  vector is `.vstack()` because we now want an  $X$  vector with two sets of distinct features.

### *Feature Scaling and Normalization function*

You have two free stories left this month.  
[Upgrade for unlimited access](#)



The code concerning *Feature Scaling and Normalization* is modified in vectors *mu* and *sigma*. Now the two vectors will accept two parameters each.

```

1  def FeatureScalingNormalization(X):
2      # Initialize the following variables:
3      # Make a copy of the X vector and call it X_norm
4      X_norm = X
5
6      # mu: It will contain the average
7      # value of X in training set.
8      mu = np.zeros(X.shape[1])
9
10     # sigma: It will contain the Range(max-min)
11     # of X or Standard Deviation
12     sigma = np.zeros(X.shape[1])
13     mu = np.vstack((X[0].mean(), X[1].mean()))
14
15     # The Standard Deviation calculation with NumPy,
16     # requires the argument "degrees of freedom" = 1
17     sigma = np.vstack((X[0].std(ddof=1), X[1].std(ddof=1)))
18
19     # number of training examples
20     m = X.shape[1]
21
22     # Make a vector of size m with the mu values
23     mu_matrix = np.multiply(np.ones(m), mu).T
24
25     # Make a vector of size m with the sigma values
26     sigma_matrix = np.multiply(np.ones(m), sigma).T
27
28     # Apply the Feature Scaling Normalization formula
29     X_norm = np.subtract(X, mu).T
30     X_norm = X_norm /sigma.T
31
32     return [X_norm, mu, sigma]
```

Code 25: The FeatureScalingNormalization function for Linear Regression with multiple variables.

## Adding a column of “ones” to the vector *X*

The line for adding “ones” to the *X* vector is modified as follows:

You have two free stories left this month.  
[Upgrade for unlimited access](#)



Code 24: Adding a column of "ones" to the vector X.

## The Prediction Query

The query for the predictions and the code for the normalization are modified as follows:

```
1 # Perform prediction basing on multiple Query
2 # Predict the Systolic Blood Pressure (SBP) for
3 # a 75 years-old human with weight = 96 Kg:
4 query = [1, 75, 96]
5
6 # Scale and Normalize the query
7 query_Normalized = [1, ((query[1]-float(mu[0]))/float(sigma[0])), ((query[2]-float(mu[1]))/float(sigma[1]))]
8
9 # Predict the SBP: (the prediction is the product of the vector [1, 75, 96] and theta)
10 prediction = np.sum(np.multiply(query_Normalized, theta))
```

PredictSBPMultiple\_1.py hosted with ❤ by GitHub

[view raw](#)

Code 25: The Query in Linear Regression with Multiple Variables

The prediction result, in this case, is an  $SBP = 143.47$

With these changes, the Python code is ready for Linear Regression with multiple variables. You have to update the code every time you will add new features from your training set!

I hope you find this post useful!

## References

1. [Andrew NG, Machine Learning | Coursera.](#)
2. John Pezzullo, Biostatistics For Dummies, Wiley, ISBN-13: 9781118553985
3. Schneider, A; Hommel, G; Blettner, M. Linear Regression Analysis Part 14 of a Series on Evaluation of Scientific Publications, Dtsch Arztebl Int 2010; 107(44): 776–82; DOI: 10.3238/arztebl.2010.0776
4. Chris Albon, Machine Learning with Python Cookbook, O'Really, ISBN-13: 978-1-492-19603-6

You have two free stories left this month.

[Upgrade for unlimited access](#)



5. Aurélien Géron, Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, O'Reilly, ISBN-13: 978-1491962299.

[Machine Learning](#)[Data Science](#)[Biomedical](#)[Linear Regression](#)[ML For Bio Data](#)

# Medium

[About](#) [Help](#) [Legal](#)

Get the Medium app



You have two free stories left this month.  
[Upgrade for unlimited access](#)

