**Simple Linear Regression**

We discussed that Linear Regression is a simple model. Simple Linear Regression is the simplest model in machine learning.

**Model Representation**

In this problem we have an input variable - **X** and one output variable - **Y**. And we want to build linear relationship between these variables. Here the input variable is called **Independent Variable**and the output variable is called **Dependent Variable**. We can define this linear relationship as follows:

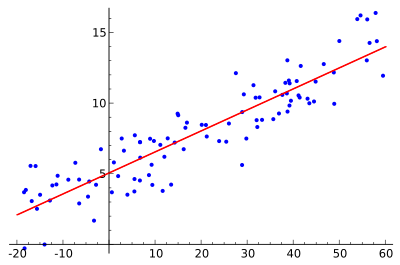
Y=β0+β1XY=β0+β1X

The β1β1 is called a scale factor or **coefficient** and β0β0 is called **bias coefficient**. The bias coeffient gives an extra degree of freedom to this model. This equation is similar to the line equation y=mx+by=mx+b with m=β1m=β1(Slope) and b=β0b=β0(Intercept). So in this Simple Linear Regression model we want to draw a line between X and Y which estimates the relationship between X and Y.

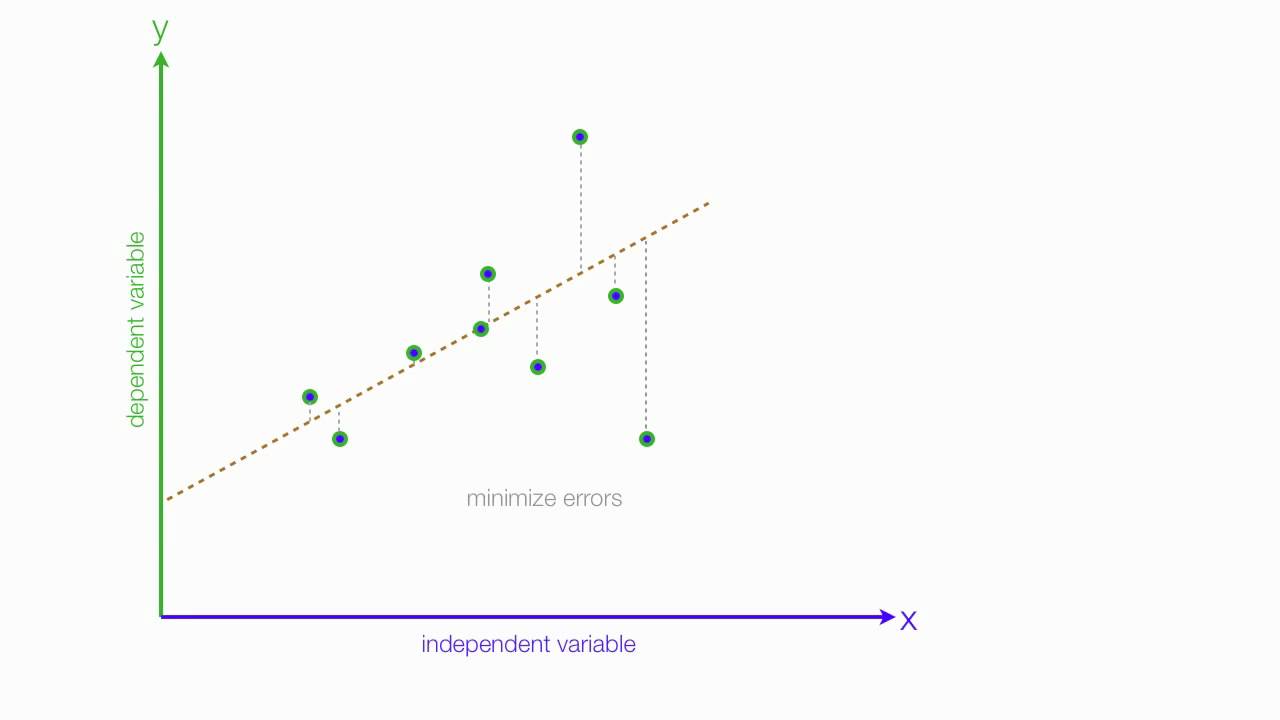
But how do we find these coefficients? That’s the learning procedure. We can find these using different approaches. One is called **Ordinary Least Square Method** and other one is called **Gradient Descent Approach**. We will use Ordinary Least Square Method in Simple Linear Regression and Gradient Descent Approach in Multiple Linear Regression in post.

**Ordinary Least Square Method**

Earlier in this post we discussed that we are going to approximate the relationship between X and Y to a line. Let’s say we have few inputs and outputs. And we plot these scatter points in 2D space, we will get something like the following image.



And you can see a line in the image. That’s what we are going to accomplish. And we want to minimize the error of our model. A good model will always have least error. We can find this line by reducing the error. The error of each point is the distance between line and that point. This is illustrated as follows.



And total error of this model is the sum of all errors of each point. ie.

D=∑i=1md2iD=∑i=1mdi2

didi - Distance between line and ith point.

mm - Total number of points

You might have noticed that we are squaring each of the distances. This is because, some points will be above the line and some points will be below the line. We can minimize the error in the model by minimizing DD. And after the mathematics of minimizing DD, we will get;

β1=∑mi=1(xi−x¯)(yi−y¯)∑mi=1(xi−x¯)2β1=∑i=1m(xi−x¯)(yi−y¯)∑i=1m(xi−x¯)2

β0=y¯−β1x¯β0=y¯−β1x¯

In these equations x¯x¯ is the mean value of input variable **X** and y¯y¯ is the mean value of output variable **Y**.

Now we have the model. This method is called [**Ordinary Least Square Method**](https://www.wikiwand.com/en/Ordinary_least_squares). Now we will implement this model in Python.

Y=β0+β1XY=β0+β1X

β1=∑mi=1(xi−x¯)(yi−y¯)∑mi=1(xi−x¯)2β1=∑i=1m(xi−x¯)(yi−y¯)∑i=1m(xi−x¯)2

β0=y¯−β1x¯β0=y¯−β1x¯

**Implementation**

We are going to use a dataset containing head size and brain weight of different people. This data set has other features. But, we will not use them in this model.. This dataset is available in this [Github Repo](https://github.com/mubaris/potential-enigma). Let’s start off by importing the data.

*# Importing Necessary Libraries*

**%**matplotlib inline

**import** numpy **as** np

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

plt**.**rcParams['figure.figsize'] **=** (20.0, 10.0)

*# Reading Data*

data **=** pd**.**read\_csv('headbrain.csv')

**print**(data**.**shape)

data**.**head(2)

(237, 4)

|  | **Gender** | **Age Range** | **Head Size(cm^3)** | **Brain Weight(grams)** |
| --- | --- | --- | --- | --- |
| **0** | 1 | 1 | 4512 | 1530 |
| **1** | 1 | 1 | 3738 | 1297 |
| **2** | 1 | 1 | 4261 | 1335 |
| **3** | 1 | 1 | 3777 | 1282 |
| **4** | 1 | 1 | 4177 | 1590 |

As you can see there are 237 values in the training set. We will find a linear relationship between Head Size and Brain Weights. So, now we will get these variables.

*# Collecting X and Y*

X **=** data['Head Size(cm^3)']**.**values

Y **=** data['Brain Weight(grams)']**.**values

To find the values β1β1 and β0β0, we will need mean of **X** and **Y**. We will find these and the coeffients.

*# Mean X and Y*

mean\_x **=** np**.**mean(X)

mean\_y **=** np**.**mean(Y)

*# Total number of values*

m **=** len(X)

*# Using the formula to calculate b1 and b2*

numer **=** 0

denom **=** 0

**for** i **in** range(m):

numer **+=** (X[i] **-** mean\_x) **\*** (Y[i] **-** mean\_y)

denom **+=** (X[i] **-** mean\_x) **\*\*** 2

b1 **=** numer **/** denom

b0 **=** mean\_y **-** (b1 **\*** mean\_x)

*# Print coefficients*

**print**(b1, b0)

0.263429339489 325.573421049

There we have our coefficients.

BrainWeight=325.573421049+0.263429339489∗HeadSizeBrainWeight=325.573421049+0.263429339489∗HeadSize

That is our linear model.

Now we will see this graphically.

*# Plotting Values and Regression Line*

max\_x **=** np**.**max(X) **+** 100

min\_x **=** np**.**min(X) **-** 100

*# Calculating line values x and y*

x **=** np**.**linspace(min\_x, max\_x, 1000)

y **=** b0 **+** b1 **\*** x

*# Ploting Line*

plt**.**plot(x, y, color**=**'#58b970', label**=**'Regression Line')

*# Ploting Scatter Points*

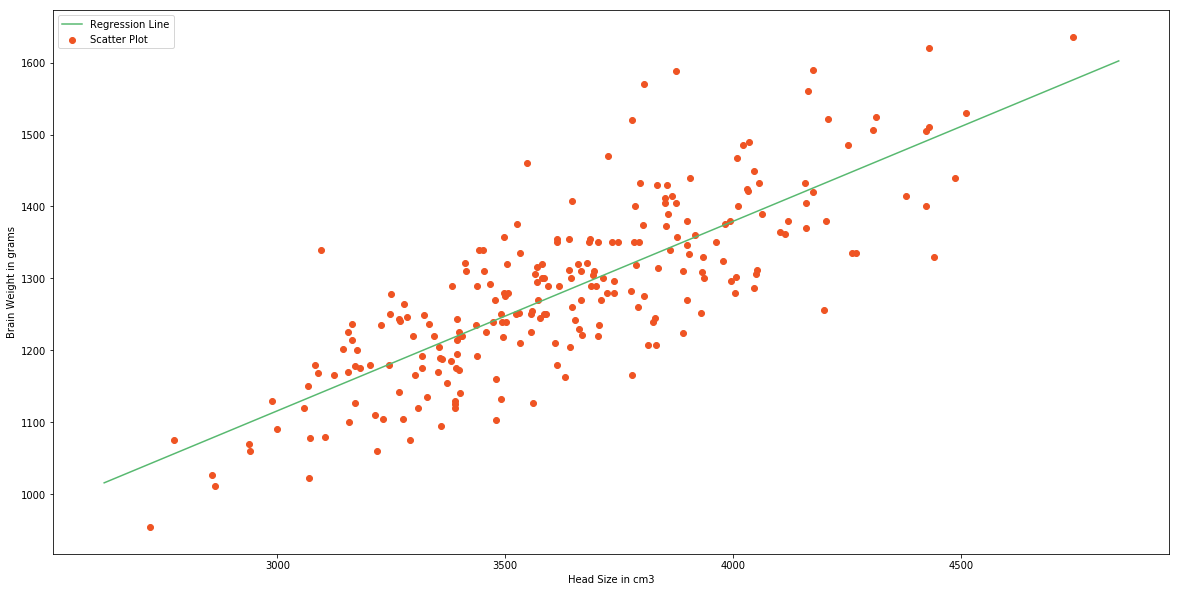
plt**.**scatter(X, Y, c**=**'#ef5423', label**=**'Scatter Plot')

plt**.**xlabel('Head Size in cm3')

plt**.**ylabel('Brain Weight in grams')

plt**.**legend()

plt**.**show()



This model is not so bad. But we need to find how good is our model. There are many methods to evaluate models. We will use **Root Mean Squared Error** and **Coefficient of Determination(**R2R2**Score)**.

Root Mean Squared Error is the square root of sum of all errors divided by number of values, or Mathematically,

RMSE=∑i=1m1m(yi^−yi)2−−−−−−−−−−−−−√RMSE=∑i=1m1m(yi^−yi)2

Here yi^yi^ is the ith predicted output values. Now we will find RMSE.

*# Calculating Root Mean Squares Error*

rmse **=** 0

**for** i **in** range(m):

y\_pred **=** b0 **+** b1 **\*** X[i]

rmse **+=** (Y[i] **-** y\_pred) **\*\*** 2

rmse **=** np**.**sqrt(rmse**/**m)

**print**(rmse)

72.1206213784

Now we will find R2R2 score. R2R2 is defined as follows,

SSt=∑i=1m(yi−y¯)2SSt=∑i=1m(yi−y¯)2

SSr=∑i=1m(yi−yi^)2SSr=∑i=1m(yi−yi^)2

R2≡1−SSrSStR2≡1−SSrSSt

SStSSt is the total sum of squares and SSrSSr is the total sum of squares of residuals.

R2R2 Score usually range from 0 to 1. It will also become negative if the model is completely wrong. Now we will find R2R2 Score.

ss\_t **=** 0

ss\_r **=** 0

**for** i **in** range(m):

y\_pred **=** b0 **+** b1 **\*** X[i]

ss\_t **+=** (Y[i] **-** mean\_y) **\*\*** 2

ss\_r **+=** (Y[i] **-** y\_pred) **\*\*** 2

r2 **=** 1 **-** (ss\_r**/**ss\_t)

**print**(r2)

0.639311719957

0.63 is not so bad. Now we have implemented Simple Linear Regression Model using Ordinary Least Square Method. Now we will see how to implement the same model using a Machine Learning Library called [scikit-learn](http://scikit-learn.org/)

**The scikit-learn approach**

[scikit-learn](http://scikit-learn.org/) is simple machine learning library in Python. Building Machine Learning models are very easy using scikit-learn. Let’s see how we can build this Simple Linear Regression Model using scikit-learn.

**from** sklearn.linear\_model **import** LinearRegression

**from** sklearn.metrics **import** mean\_squared\_error

*# Cannot use Rank 1 matrix in scikit learn*

X **=** X**.**reshape((m, 1))

*# Creating Model*

reg **=** LinearRegression()

*# Fitting training data*

reg **=** reg**.**fit(X, Y)

*# Y Prediction*

Y\_pred **=** reg**.**predict(X)

*# Calculating RMSE and R2 Score*

mse **=** mean\_squared\_error(Y, Y\_pred)

rmse **=** np**.**sqrt(mse)

r2\_score **=** reg**.**score(X, Y)

**print**(np**.**sqrt(mse))

**print**(r2\_score)

72.1206213784

0.639311719957

You can see that this exactly equal to model we built from scratch, but simpler and less code.

Now we will move on to Multiple Linear Regression.

**Multiple Linear Regression**

Multiple Linear Regression is a type of Linear Regression when the input has multiple features(variables).

**Model Representation**

Similar to Simple Linear Regression, we have input variable(**X**) and output variable(**Y**). But the input variable has nn features. Therefore, we can represent this linear model as follows;

Y=β0+β1x1+β1x2+…+βnxnY=β0+β1x1+β1x2+…+βnxn

xixi is the ith feature in input variable. By introducing x0=1x0=1, we can rewrite this equation.

Y=β0x0+β1x1+β1x2+…+βnxnY=β0x0+β1x1+β1x2+…+βnxn

x0=1x0=1

Now we can convert this eqaution to matrix form.

Y=βTXY=βTX

Where,

β=[β0β1β2..βn]Tβ=[β0β1β2..βn]T

and

X=[x0x1x2..xn]TX=[x0x1x2..xn]T

We have to define the cost of the model. Cost bascially gives the error in our model. **Y** in above equation is the our hypothesis(approximation). We are going to define it as our hypothesis function.

hβ(x)=βTxhβ(x)=βTx

And the cost is,

J(β)=12m∑i=1m(hβ(x(i))−y(i))2J(β)=12m∑i=1m(hβ(x(i))−y(i))2

By minimizing this cost function, we can get find ββ. We use **Gradient Descent** for this.

**Gradient Descent**

Gradient Descent is an optimization algorithm. We will optimize our cost function using Gradient Descent Algorithm.

**Step 1**

Initialize values β0β0, β1β1,…, βnβn with some value. In this case we will initialize with 0.

**Step 2**

Iteratively update,

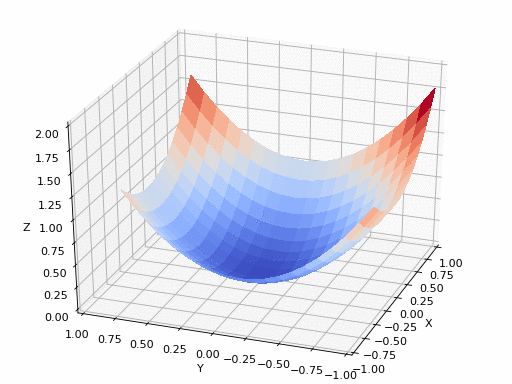
βj:=βj−α∂∂βjJ(β)βj:=βj−α∂∂βjJ(β)

until it converges.

This is the procedure. Here αα is the learning rate. This operation ∂∂βjJ(β)∂∂βjJ(β) means we are finding partial derivate of cost with respect to each βjβj. This is called Gradient.

Read [this](https://math.stackexchange.com/questions/174270/what-exactly-is-the-difference-between-a-derivative-and-a-total-derivative) if you are unfamiliar with partial derivatives.

In step 2 we are changing the values of βjβj in a direction in which it reduces our cost function. And Gradient gives the direction in which we want to move. Finally we will reach the minima of our cost function. But we don’t want to change values of βjβj drastically, because we might miss the minima. That’s why we need learning rate.



The above animation illustrates the Gradient Descent method.

But we still didn’t find the value of ∂∂βjJ(β)∂∂βjJ(β). After we applying the mathematics. The step 2 becomes.

βj:=βj−α1m∑i=1m(hβ(x(i))−y(i))x(i)jβj:=βj−α1m∑i=1m(hβ(x(i))−y(i))xj(i)

We iteratively change values of βjβj according to above equation. This particular method is called **Batch Gradient Descent**.

**Implementation**

Let’s try to implement this in Python. This looks like a long procedure. But the implementation is comparitively easy since we will vectorize all the equations. If you are unfamiliar with vectorization, read this [post](https://www.datascience.com/blog/straightening-loops-how-to-vectorize-data-aggregation-with-pandas-and-numpy/)

We will be using a student score dataset. In this particular dataset, we have math, reading and writing exam scores of 1000 students. We will try to find a predict the score of writing exam from math and reading scores. You can get this dataset from this [Github Repo](https://github.com/mubaris/potential-enigma). That’s we have 2 features(input variables). Let’s start by importing our dataset.

**%**matplotlib inline

**import** numpy **as** np

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

plt**.**rcParams['figure.figsize'] **=** (20.0, 10.0)

**from** mpl\_toolkits.mplot3d **import** Axes3D

data **=** pd**.**read\_csv('student.csv')

**print**(data**.**shape)

data**.**head()

(1000, 3)

|  | **Math** | **Reading** | **Writing** |
| --- | --- | --- | --- |
| **0** | 48 | 68 | 63 |
| **1** | 62 | 81 | 72 |
| **2** | 79 | 80 | 78 |
| **3** | 76 | 83 | 79 |
| **4** | 59 | 64 | 62 |

We will get scores to an array.

math **=** data['Math']**.**values

read **=** data['Reading']**.**values

write **=** data['Writing']**.**values

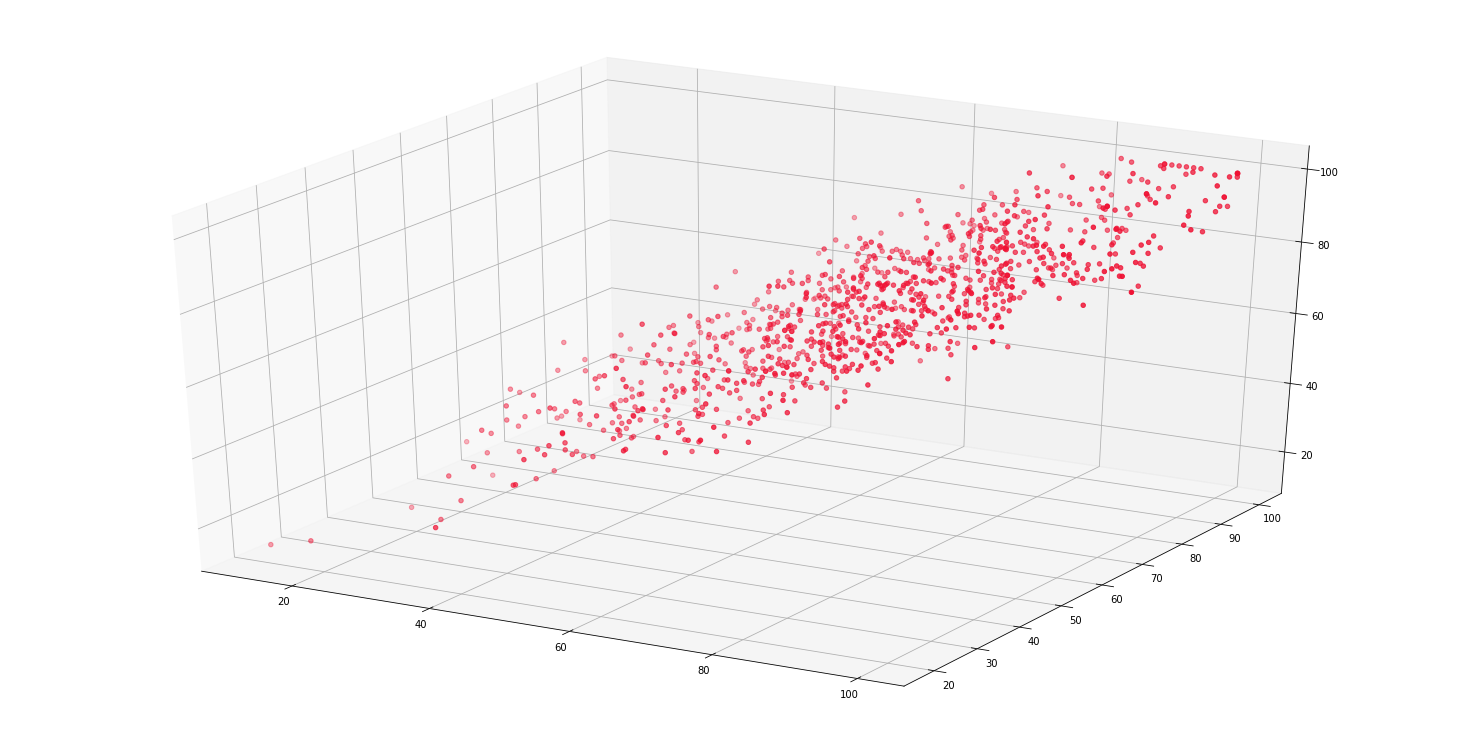
*# Ploting the scores as scatter plot*

fig **=** plt**.**figure()

ax **=** Axes3D(fig)

ax**.**scatter(math, read, write, color**=**'#ef1234')

plt**.**show()



Now we will generate our X, Y and ββ.

m **=** len(math)

x0 **=** np**.**ones(m)

X **=** np**.**array([x0, math, read])**.**T

*# Initial Coefficients*

B **=** np**.**array([0, 0, 0])

Y **=** np**.**array(write)

alpha **=** 0.0001

We’ll define our cost function.

**def** **cost\_function**(X, Y, B):

m **=** len(Y)

J **=** np**.**sum((X**.**dot(B) **-** Y) **\*\*** 2)**/**(2 **\*** m)

**return** J

inital\_cost **=** cost\_function(X, Y, B)

**print**(inital\_cost)

2470.11

As you can see our initial cost is huge. Now we’ll reduce our cost prediocally using Gradient Descent.

**Hypothesis:**hβ(x)=βTxhβ(x)=βTx

**Loss:**(hβ(x)−y)(hβ(x)−y)

**Gradient:**(hβ(x)−y)xj(hβ(x)−y)xj

**Gradient Descent Updation:**βj:=βj−α(hβ(x)−y)xj)βj:=βj−α(hβ(x)−y)xj)

**def** **gradient\_descent**(X, Y, B, alpha, iterations):

cost\_history **=** [0] **\*** iterations

m **=** len(Y)

**for** iteration **in** range(iterations):

*# Hypothesis Values*

h **=** X**.**dot(B)

*# Difference b/w Hypothesis and Actual Y*

loss **=** h **-** Y

*# Gradient Calculation*

gradient **=** X**.**T**.**dot(loss) **/** m

*# Changing Values of B using Gradient*

B **=** B **-** alpha **\*** gradient

*# New Cost Value*

cost **=** cost\_function(X, Y, B)

cost\_history[iteration] **=** cost

**return** B, cost\_history

Now we will compute final value of ββ

*# 100000 Iterations*

newB, cost\_history **=** gradient\_descent(X, Y, B, alpha, 100000)

*# New Values of B*

**print**(newB)

*# Final Cost of new B*

**print**(cost\_history[**-**1])

[-0.47889172 0.09137252 0.90144884]

10.4751234735

We can say that in this model,

Swriting=−0.47889172+0.09137252∗Smath+0.90144884∗SreadingSwriting=−0.47889172+0.09137252∗Smath+0.90144884∗Sreading

There we have final hypothesis function of our model. Let’s calculate **RMSE** and R2R2**Score** of our model to evaluate.

*# Model Evaluation - RMSE*

**def** **rmse**(Y, Y\_pred):

rmse **=** np**.**sqrt(sum((Y **-** Y\_pred) **\*\*** 2) **/** len(Y))

**return** rmse

*# Model Evaluation - R2 Score*

**def** **r2\_score**(Y, Y\_pred):

mean\_y **=** np**.**mean(Y)

ss\_tot **=** sum((Y **-** mean\_y) **\*\*** 2)

ss\_res **=** sum((Y **-** Y\_pred) **\*\*** 2)

r2 **=** 1 **-** (ss\_res **/** ss\_tot)

**return** r2

Y\_pred **=** X**.**dot(newB)

**print**(rmse(Y, Y\_pred))

**print**(r2\_score(Y, Y\_pred))

4.57714397273

0.909722327306

We have very low value of RMSE score and a good R2R2 score. I guess our model was pretty good.

Now we will implement this model using scikit-learn.

**The scikit-learn Approach**

scikit-learn approach is very similar to Simple Linear Regression Model and simple too. Let’s implement this.

**from** sklearn.linear\_model **import** LinearRegression

**from** sklearn.metrics **import** mean\_squared\_error

*# X and Y Values*

X **=** np**.**array([math, read])**.**T

Y **=** np**.**array(write)

*# Model Intialization*

reg **=** LinearRegression()

*# Data Fitting*

reg **=** reg**.**fit(X, Y)

*# Y Prediction*

Y\_pred **=** reg**.**predict(X)

*# Model Evaluation*

rmse **=** np**.**sqrt(mean\_squared\_error(Y, Y\_pred))

r2 **=** reg**.**score(X, Y)

**print**(rmse)

**print**(r2)

4.57288705184

0.909890172672