Converting Business Problems to Data Problems

At the end of the day; whatever we are doing here is about money. To put that more appropriately , we are developing these techniques to solve business problems. Real business problems do not come conveniently all dressed up as a data problem in general. For example consider this :

- A bank is making too many losses because of defaulters on retail loans

Its a genuine business problem but it isn't a data problem yet on the face of it. So, what is a data problem then. A data problem is a business problem expressed in terms of the data [potentially]available in the business process pipeline.

It majorly has two components:

- Response/Goal/Outcome
- Set of factor/features data which affects our goal/response/outcome

Lets look at the loan default problem and find these components.

- Outcome is loan default, which we would like to predict when considering giving loan to a prospect.
- What factors could help us in doing that. Banks collect a lot of information on a loan application such as financial data, personal information. In addition to that they also make queries regarding credit history to various agencies. We could use all these features/information to predict whether a customer is going to default on their loan or not and then reconsider our decision of granting loans depending on the result.

Here are few more business problems which you can try converting to data problems :

```
* A marketing campaign is causing spam complaints from the existing customers

* Hospitals can not afford to have additional staff all year round
which are needed only when patient intakes become higher than a certain amount

* An ecommerce company wants to know how it should plan for the budget on cloud servers

* How many different election campaign strategies should a party opt for

* What kind of toys should lego launch in India
```

Three kinds of Problems

If you went through the business problems mentioned above, you'd have realized there are mainly three kind of problems which can then be clubbed into two categories :

1. Supervised
2. Unsupervised

Supervised problems are the problems which have explicit outcomes. Such as default on loan, Required Number of Staff, Server Load etc. Within these , you can see separate kinds.

1. Regression
2. Classification

Regression problems are those where outcome is a continuous numeric value e.g. Sales , Rainfall , Server Load (values over a range with technically infinite unique values possible as outcome and have an ordinal relationship [e.g. 100 is twice as much as to 50]).

Classification problem on the other hand have their outcome as categories [e.g.: good/bad/worse; yes/no ; 1/0 etc], with limited number of defined outcomes .

Unsupervised problems are those where there is no explicit measurable outcome associated with the problem. You just need to find general pattern hidden in the measured factors. Finding different customer segments or electoral segments or finding latent factors in the data comes under such problems.

Now categories of problems can be formally grouped like this:

1. Supervised
    1. Regression
    2. Classification
2. Unsupervised

Our focus in this module will be over **Supervised** problems with an existing outcome which we are trying to predict in context of a regression or a classification problem

Data Problem to Mathematical Problem

Lets discuss key takeaways from the discussion above; which will help us in converting a data problem (of supervised kind) to its mathematical representation.

- We want to make use of historical data to extract pattern so that we can build a predictive model for future/new data
- We want our solution to be as accurate as possible

Now, what do we mean by pattern ? In mathematical terms; we are looking for a function which takes input (the values of factors affecting my response/outcome) and outputs the value of outcome (which we call predictions)

Regression Problem

We'll denote our prediction for i^{th} observation as $\hat{y}_i$ and real value of the outcome in the historical data given to us as y_i . And this function that we talk about is denoted by `f` . Inputs collectively are denoted as X_i

$\hat{y_i} \ = \ f(X_i)$

It isn't really possible to have a function which will make perfect predictions [in fact it is possible but not good to have a function which makes perfect prediction, its called overfitting ; we'll keep that discussion for later] .

Our predictions are going to have errors . We can calculate those errors easily by comparing our predictions with real outcomes .

$e_i \ = \ y_i \ - \ \hat{y_i}$

We would want this error to be as small as possible across all observations. One way to represent the error `across all observation` will be average error for the entire data . However simple average is going to be meaning less , because errors for different observations might have different signs; some negative some positive. Overall average error might as well be zero, but that doesnt mean individual errors dont exist. We need to come up with something for this error `across all observation` which doesnt consider sign of errors . There are couple of ideas that we can consider .

- Mean Absolute Error $\sum_i^{i=n} |e_i|$
- Mean Squared Error $\sum_i^{i=n} {e_i}^2$

These here are called **Cost Functions**, another popular name for the same is Loss Function . They are also mentioned as just Cost/Loss . In many places , these are considered as averages but simple sum [Sum of absolute errors or sum of squared errors] . It doesnt really matter because difference between them is division by a constant (number of observations); it doesnt affect our pattern extraction or function estimation for prediction.

Among the two that we mentioned here , Mean Squared Error is more popular in comaprison to Mean Absolute Error due to its easy differentiability. How does that matter? It'll be clear once we discuss `Gradient Descent`.

Lets take a pause here and understand , what do these cost functions represent? These represent our business expectation of model being as accurate as possible in a mathematically quantifiable way .

We would want our prediction function $f(X_i)$ to be such that , for the given historical data, Mean Squared Error (or whatever other cost function you are considering) is as small as possible .

Classification Problem

It was pretty straight forward, as to what do we want to predict in Regression Problem. However it isnt that simple for classification problem as you might expect . Consider the following data on customer's Age and their response to a product campaign for an insurance policy.

Age	Outcome	Age	Outcome
20	NO	30	NO
20	NO	30	NO
20	NO	30	YES
20	NO	30	YES
20	YES	30	YES
25	NO	35	NO
25	NO	35	YES
25	NO	35	YES
25	YES	35	YES
25	YES	35	YES

If I asked you , what will be the outcome if somebody's age is 31, according to the data shown above . Your likely answer is `YES` as you see that majority of the people with increasing age have said `YES`.

If I further asked you whats the outcome for somebody with age 34, your response will again be `YES` . Now, whats the difference between these two cases . You can easily see that outcome is **more likely** to be `YES` when the age is higher . This tells us that , we are not really interested in absolute predictions `YES/NO`; but in the probability of the outcome being `YES/NO` for given inputs.

More formally we are interested in this : $P(y_i = YES|X_i)$

`YES/NO` at the end of the day are just labels. We'll consider them to be 1/0 to make our life mathematically easy; as will be evident in some time. Also to keep the notation concise , we'll use just p_i instead of $P(y_i = 1|X_i)$

Now that we have figure out that we want to predict p_i, Lets see how do we write that against our prediction function $f(X_i)$

range of p_i being a probability is between 0 to 1. However $f(X_i)$ theoretically can be anything between $-\infty$ to $+\infty$. it wont make sense to write $p_i = f(X_i)$

we need to apply some transformation on any one side to ensure that the ranges match. sigmoid or logit function is one such transformation which is popular in use.

$p_i = \frac{1}{1+e^{-f(X_i)}}$

another format [just rearranged terms] for the same is

$log(\frac{p_i}{1-p_i}) = f(X_i)$

Finally, we are clear about what we want to predict and how it is written against our prediction function . We can now start discussing what are our business expectation from this and how do we represent the same in form of a cost/loss function .

We'll ideally want the probabiility of outcome being 1 to 100% when outcome in reality is 1 , and probability of outcome being 1 to be 0% when outcome in reality is 0 . But, as usual , perfect solution is practically not possible. Closest compromise will be that p_i is as close to 1 as possible when outcome in reality is 1 and it is as close to 0 as possible when outcome in reality is 0.

$$p_i \uparrow \qquad \forall \qquad y_i = 1$$
$$p_i \downarrow \qquad \forall \qquad y_i = 0$$

since p_i takes value in the range [0,1] , we can say that when p_i goes close to 0, it implies that $1 - p_i$ goes close to 1. So the above expression can be written as :

$$p_i \uparrow \qquad \forall \qquad y_i = 1$$
$$(1 - p_i) \uparrow \qquad \forall \qquad y_i = 0$$

This is still at intuition level, we need to find a way to convert this idea into a mathematical expression which can be used as a cost function . Consider this expression :

$$L_i = {p_i}^{y_i} (1 - p_i)^{(1-y_i)}$$

Expression L_i is known as likelihood because it gives you the probability of the real outcome that you get. This expression takes value p_i when $y_i = 1$ and takes value $(1 - p_i)$ when $y_i = 0$. Using the idea that we devised above, we can rewrite it like this :

$$L_i \uparrow \qquad \forall \qquad y_i = 1$$
$$L_i \uparrow \qquad \forall \qquad y_i = 0$$

This implies that we want the likelihood to be as high as possible irrespective of what the outcome (1/0) is. Another way to express that is , that we want likelihood to align with whatever the real outcome is . As earlier we are not concerned with likelihood of a single observation, we are interested in it at over all level. Since likelihood is nothin but a probability , if we want to calculate likehood of entire data, it'll be nothing but multiplication of all individual likelihoods.

$$L = \prod_{i=1}^{i=n} L_i$$

This is our cost function which we need to maximise . we want such an $f(X_i)$ for which L is as high possible for the given historical data. However this is not the form in which this is used . Since optimising a quantity where individual terms are multiplied with each other is a very difficult problem to solve we'll instead use log(L) as our cost function , which makes individual terms getting summed up instead .

$$log(L) = \sum_{i=1}^{i=n} L_i$$

To make this a standard minimisation problem , instead of maximising log(L), minimisation of -log(L) is considered

Finally , cost function for classification problem is -log(L) , also known as **negative log likelihood**

other forms of the same which you'll get to see :

$$-log(L) = -\sum_{i=1}^{i=n} [y_i log(p_i) + (1 - y_i) log(1 - p_i)]$$

this can also be written in terms of prediction function as follows :

$-log(L) = -\sum_{i=1}^{i=n}[y_i f(X_i) - log(1+e^{f(X_i)})]$

Estimating $f(X_i)$

For this discussion we'll consider regression with $f(X_i)$ as a linear model , but the same idea will be applicable to any parametric model

Lets say we are trying to predict sales of a jewelery shop, considering gold price that day and temp that day as factors affecting sales . We can consider a linear model like this

$sales_i = \beta_0 + \beta_1 * price_i + \beta_2 * temp_i$

Here βs are some constants. How do we determine, what values of βs should we consider ?

As per the discussion that we have had about business expectation , we would desire βs for which the cost function is as low as possible for the given historical data given to us .

We can try out many different values of βs and select the ones for which our cost function is coming out to be as low as possible. One way to find good βs can be to simply compare the values of cost functions for each and pick the one with lowest cost function value.

Here are few steps of the process :

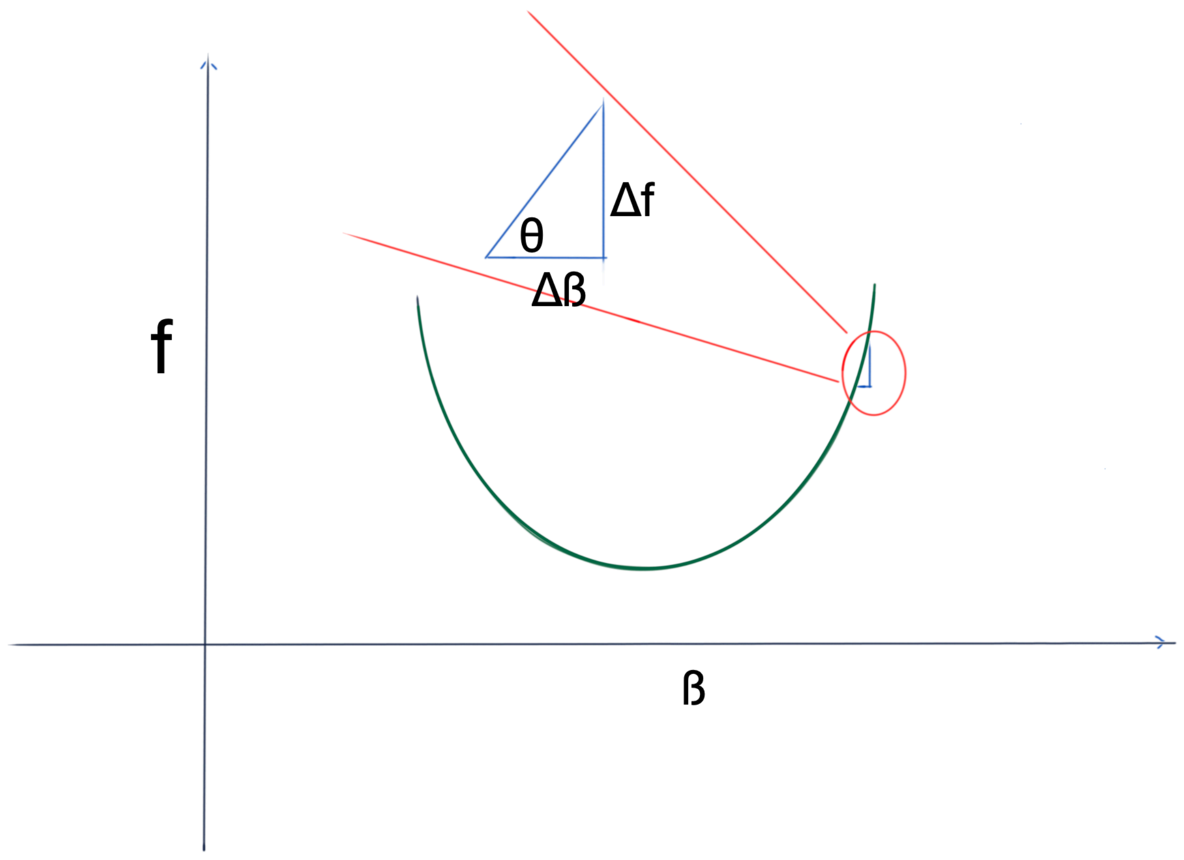
β_0	β_1	β_2	**Cost**	**Decision**
10	5	0	140	Start
-5	-2	4	160	Discard
1	3	7	150	Discard
4	7	02	135	Switch to theses Values
2	-2	10	130	Switch to theses Values
1	4	6	132	Discard
3	6	-5	141	Discard

We can keep on trying random values of βs like that and swtitching to new values whenever we encounter a combination which gives us a new low for the cost function.

And we can stop randomly trying new values of βs if we dont get any lower value for cost function for a long time .

This works in theory , given ; that we have infinite amount of time, resources and most important; patience!. We would want to have another method, which enables us to change our βs in a such a way that cost function always goes down [instead of changing randomly and discarding most of them] .

Gradient Descent



Here in this image we can see that f is function with parameter β . If we change β by a small amount $\Delta\beta$, function changes by Δf . As long as we ensure that $\Delta\beta$ is small , we can assume the triangle shown with sides $\Delta\beta$ and Δf, is right angle triangle , and we can write :

$$tan(\theta) = \frac{\Delta f}{\Delta\beta} \qquad (1)$$

For very small $\Delta\beta$ we can write $tan(\theta)$ in terms of gradient as follows:

$$tan(\theta) \ = \frac{\delta f}{\delta\beta} \qquad (2)$$

Equating this to (1) and after doing a little adjustment we get:

$$\Delta f \ = \ \frac{\delta f}{\delta\beta}\Delta\beta \qquad (3)$$

if there were multiple paramters involved , say β_1 and β_2

then changing those parameters will individually contribute to changing the function

$$\Delta f \ = \ \frac{\delta f}{\delta\beta_1}\Delta\beta_1 \ + \ \frac{\delta f}{\delta\beta_2}\Delta\beta_2 \qquad (4)$$

this can be written as dot product between two vectors , respectively know as gradient and change in parameters

$$\nabla f \ = \ (\frac{\delta f}{\delta\beta_1}, \frac{\delta f}{\delta\beta_2})$$

$$\Delta\beta \ = \ (\Delta\beta_1, \Delta\beta_2)$$

(4) can be re written as :

$$\Delta f \ = \ \nabla f \cdot \Delta\beta \qquad (5)$$

This is not some expression that you havent seen before , in fact we use it in our daily lives all the time. This expression simply means that , if we change our parameter by some amount , change in function can be calculated by multiplying change in parameter with gradient of the function w.r.t. the parameter.

Where do we use that in real life ?

When somebody asks you to find how much distance you covered if you were travelling at 4km/minute for 20 minutes .

you simply tell them 80km!, here distance is nothing but function of time ,

$\Delta f = 80km$ & $\Delta\beta = 20mins$, the speed is nothing but rate of change of distance w.r.t. time, or in other words , gradient of distance w.r.t. time $\nabla f = 4km/minute$ [gradient is nothing but rate of change !!]

You must be wondering , why are we talking about all of this ? eq (5) is magical . This gives us an idea about how we can change our parameters such that cost function always goes down

Consider $\Delta\beta \ = \ -\eta\nabla f \qquad\qquad (6)$

Where η is some positive constant . if we put this back in (5), lets see what happens

$$\Delta f \ = \ \nabla f \cdot \Delta\beta \ = -\eta\nabla f \cdot \nabla f = -\eta||\nabla f||^2 \qquad\qquad (7)$$

(7) is an amazing result , it tells us that if we changed our paramter , as per the suggestion in (6) , change in function will always be negative . This gives us a consistent way of changing our parameters so that our cost function always goes down. Once we start to reach near the optimal value, gradient of the cost function ∇f will tend to zero and our parameter will stop to change .

Now we have consistent method for starting from random values of βs and changing them in such a way that we eventually arrive at optimal values of βs for given historical data. Lets see whether it really works or not with an example in context of linear regression .

Linear Regression Parameter Estimation example with Gradient Descent

Recall our discussion on cost function for linear regression , it looked like this :

$$SSE \ = \ \sum {e_i}^2 \ = \ \sum (y_i \ - \ \hat{y_i})^2 \ = \ \sum (\ y_i \ - \ \beta_0 \ - \ \beta_1 * x_i)^2$$

If we consider entire X ,Y data and parameters βs to be written in matrix format like this

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

$$X = \begin{bmatrix} 1 & x_{11} & x_{21} & . & . & x_{p1} \\ 1 & x_{12} & x_{22} & . & . & x_{p2} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & x_{1n} & x_{2n} & . & . & x_{pn} \end{bmatrix}$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ \beta_p \end{bmatrix}$$

We can write :

predictions = $X\beta$ [keep in mind that this will be a matrix multiplication]

errors = Y - predictions = $Y - X\beta$

cost $= (Y - X\beta)^T(Y - X\beta)$ = $errors^T error$

I have taken liberty to extend the idea to more features which you see here as part of X matrix. x_{ij} represents j^{th} value of i^{th} variables/feature. All 1 column in X matrix represent multipliers of β_0.

Lets see what is the gradient of the loss function . Keep in mind that y_i and x_{ij} here are numbers/data points from the data. They are not parameters. They are observed values of the features.

gradient = $\nabla f = -2X^T(Y - X\beta)$

Lets simulate some data and put gradient descent to test

```
import pandas as pd
import numpy as np
```

```
x1=np.random.randint(low=1,high=20,size=20000)


x2=np.random.randint(low=1,high=20,size=20000)
```

```
y=3+2*x1-4*x2+np.random.random(20000)
```

you can see that we have generated data such that y is an approximate linear combination of x1 and x2, next we'll calculate optimal parameter values using gradient descent and compare them with results from sklearn and we'll see how good is the method.

```
x=pd.DataFrame({'intercept':np.ones(x1.shape[0]),'x1':x1,'x2':x2})

w=np.random.random(x.shape[1])


```

Lets write functions for predictions, error, cost and gradient that we discussed above

```
def myprediction(features,weights):

    predictions=np.dot(features,weights)
    return(predictions)

myprediction(x,w)
```

```
array([ 3.83044239,  5.51284694,  8.82009525, ...,  4.82993684,
       10.15713504,  2.49340259])
```

Note that , `np.dot` here is being used for matrix multiplication . Simple multiplication results to element wise multiplication , which is simply wrong in this context .

```
def myerror(target,features,weights):
    error=target-myprediction(features,weights)
    return(error)
myerror(y,x,w)
```

```
array([ -8.34075913, -26.3578085 , -53.60220599, ..., -39.73305801,
       -22.3456587 , -39.18996579])
```

```
def mycost(target,features,weights):
    error=myerror(target,features,weights)
    cost=np.dot(error.T,error)
    return(cost)

mycost(y,x,w)
```

```
23139076.992828812
```

```python
def gradient(target,features,weights):

    error=myerror(target,features,weights)

    gradient=-np.dot(features.T,error)/features.shape[0]

    return(gradient)

gradient(y,x,w)
```

```
array([ 24.86385998, 212.05914008, 369.58961913])
```

Note that gradient here is vector of 3 values because there are 3 parameters . Also since this is being evaluated on the entire data, we scaled it down with number of observations . Do recall that , the approximation which led to the ultimate results was that change in parameters is small. We dont have any direct control over gradient , we can always chose a small value for η to ensure that change in parameter remains small. Also if we end up chosing too small value for η, we'll need to take larger number of steps to change in parameter in order to arrive at the optimal value of the parameters

Lets looks at the expected value for parameters from sklearn . Dont worry about the syntax here , we'll discuss that in detail, when we formally start with linear models in next module .

```python
from sklearn.linear_model import LinearRegression
```

```python
lr=LinearRegression()
lr.fit(x.iloc[:,1:],y)
sk_estimates=([lr.intercept_]+list(lr.coef_))
```

```python
sk_estimates
```

```
[3.4963871364502594, 2.0000361062367253, -3.999828135637543]
```

When you run the same , these might be different for you, as we generated the data randomly

Now lets write our version of this , using gradient descent

```
def my_lr(target,features,learning_rate,num_steps,print_when):

    # start with random values of parameters
    weights=np.random.random(features.shape[1])

    # change parameter multiple times in sequence
    # using the cost function gradient which we discussed earlier
    for i in range(num_steps):

        weights -= learning_rate*gradient(target,features,weights)

    # this simply prints the cost function value every (print_when)th iteration
        if i%print_when==0:
            print(mycost(target,features,weights),weights)

    return(weights)


```

```
my_lr(y,x,.0001,500,100)
```

```
result after 100th iteration : 29897491.344063997 [0.86957516 0.90950957 0.35280286]
result after 200th iteration : 5141497.711763546 [ 0.75555271  0.26200844 -1.71416641]
result after 300th iteration : 2703652.3603685475 [ 0.74787632  0.6560246 -2.37514143]
result after 400th iteration : 1480112.0586781118 [ 0.75044815  1.03096429 -2.77924166]
result after 500th iteration : 815173.9884761975 [ 0.75398277  1.31567619 -3.06931226]
```

```
final weights after 500 iterations: array([ 0.75755246,  1.52465372, -3.28073366])
```

you can see that if we take too few steps , we did not reach to the optimal value

Lets increase the learning rate η

```
my_lr(y,x,.01,500,100)
```

```
result after 100th iteration : 40621523.14640909 [ 0.2279513  -1.98371006
-3.60180851]
result after 200th iteration : 8.399918899418999e+30 [-8.28197980e+10
-9.56164546e+11 -9.47578310e+11]
result after 300th iteration : 2.0273152258521542e+54 [-4.06871492e+22
-4.69738039e+23 -4.65519851e+23]
result after 400th iteration : 4.892912746164989e+77 [-1.99885070e+34
-2.30769721e+35 -2.28697438e+35]
result after 500th iteration : 1.1809014620072595e+101 [-9.81981827e+45
-1.13370985e+47 -1.12352928e+47]
```

```
final weights after 500 iterations:array([3.68564341e+57, 4.25511972e+58,
4.21690928e+58])
```

You can see that because of high learning rate , change is parameter is huge and we end up missing the optimal point , cost function values , as well as parameter values ended up exploding. Now lets run with low learning rate and higher number of steps

```
my_lr(y,x,.0004,100000,10000)
```

```
result after 10000th iteration : 30786777.19050019 [0.17828571 0.48834937
0.70073856]
result after 20000th iteration : 12603.561197114073 [ 1.44895395
2.0897855  -3.91144197]
result after 30000th iteration : 5543.348390660241 [ 2.27839978
2.05342668 -3.94724853]
result after 40000th iteration : 3044.822379129174 [ 2.77182469
2.03179736 -3.96854931]
result after 50000th iteration : 2160.623502773709 [ 3.06535578
2.0189304  -3.98122083]
result after 60000th iteration : 1847.715952746199 [ 3.23997303
2.01127604 -3.98875893]
result after 70000th iteration : 1736.981662664487 [ 3.34385023
2.00672257 -3.99324323]
result after 80000th iteration : 1697.7941039148957 [ 3.40564521
2.00401379 -3.99591087]
result after 90000th iteration : 1683.926089232013 [ 3.44240612
2.00240237 -3.99749782]
result after 100000th iteration : 1679.018362458989 [ 3.46427464
2.00144376 -3.99844186]
```

```
final result after 100000th iteration :array([ 3.4772829 ,  2.00087354,
-3.99900342])
```

We can see here that we ended up getting pretty good estimates for βs , as good as from sklearn
.

```
sk_estimates
```

```
[3.4963871364502594, 2.0000361062367253, -3.999828135637543]
```

there are modifications to gradient descent in which can achieve the same thing in much less number of iterations. We'll discuss that in detail when we start with our course in Deep learning. For now ,we'll conclude this module here.