Recap

1. Simple Linear Regression: In Simple Linear Regression there are 1 input & 1 output & we assume that data is Linear means It has linear relationship.

This algorithm will draw a best fit line which passes very closely to each data point & will predict the y-axis value correspnding to the x-axis.

Eqn: y = mx + c, where m is the slop or angle or how much y will change when x is changed & called coefficient. b is intercept.

there are 2 ways to find the values of m & b..

1. OLS method: frmulization, computatinally expensive
2. Gradient Descent: Approximating, arrives very close to correct value

2. Regression Metrics MAE, MSE, RMSE, r2-score, adjusted r2 score

3. Multiple Linear Regression: When there are more then 1 input cols, In SLR, It creates best fit line.. but in MLR it creates Hyperplane which passes very close to all the points in 3d or nd environment.

y = b0 + b1x1 + b2x2... bnxn

our goal in SLR & MLR is to minimize the loss function & increase r2 score

Above ways of cpmputing b value is OLS method but its time consuming, if we have large data then it will take a lot of time thats why we can use approximation technique Gradient descent

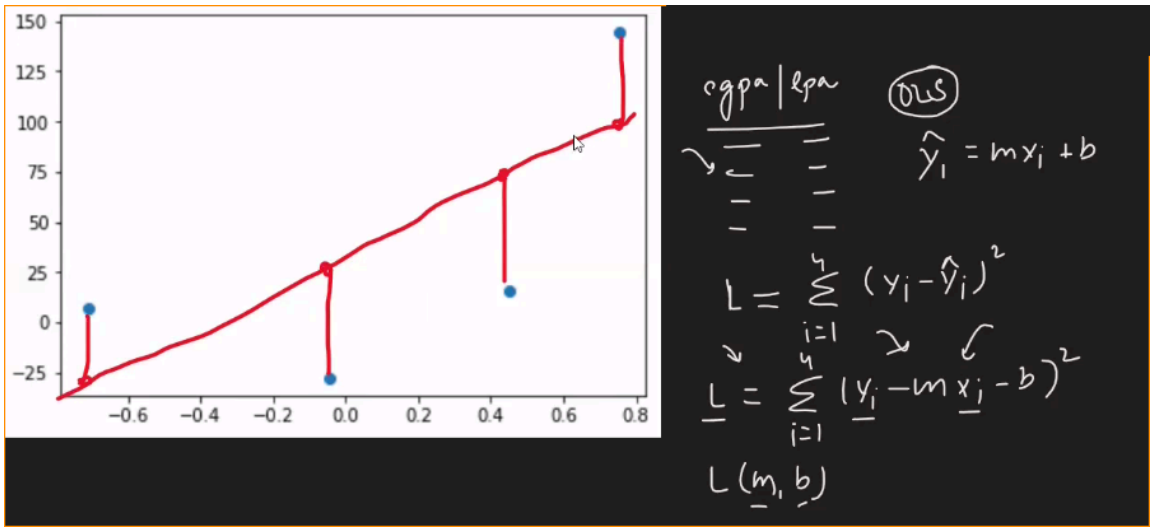
4. Gradient Descent: In gradient descent we try to minimize the loss function for b value is minimum & it wll be minimum where slope == 0. so we initialize with a random b value & move to opposite direction of where slope is increasing & eventually old slope & new slope becomes equal so we arrive at minimum point which is the point where loss function is minimum.

Gradient Descent

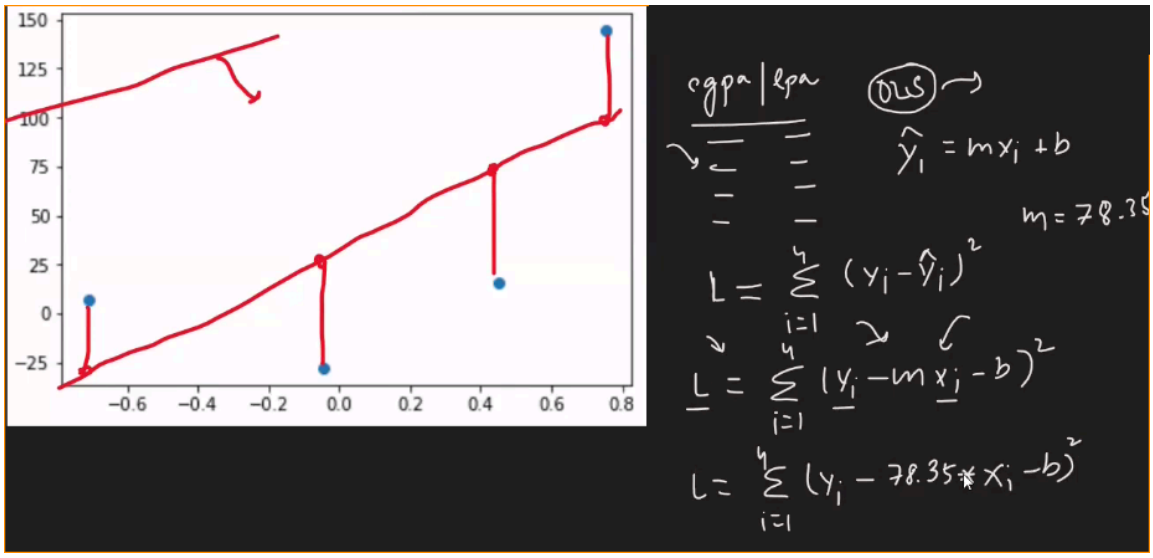
- Gradient descent is first order iterative optimizing algorithm for finding a local minima of a diffrentiable function
- The idea is to take repetative steps in the opposite direction of the gradient of the function at the current point
- eventually reaching local minima
- Its used in Linear, logistic, TSNE, complete deep learning etc

Intuition

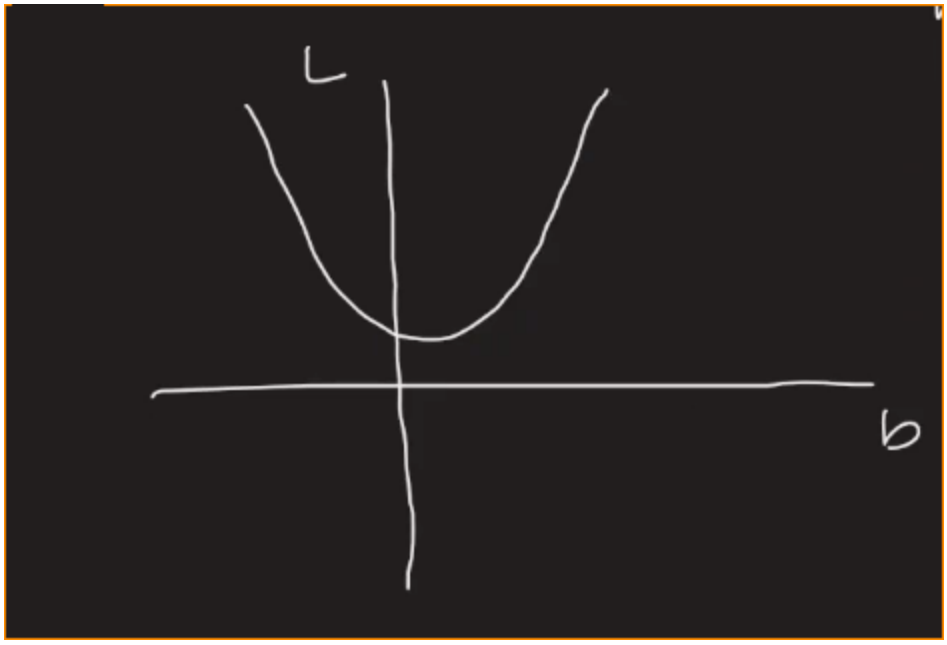
- This is the data of cgpa lpa of4 students
- I've to draw a best fit line which passes very closely to all points means making very less error b/w actual & predicted using gradient descent
- means we have to find the value of such m & b for which error will be minimum



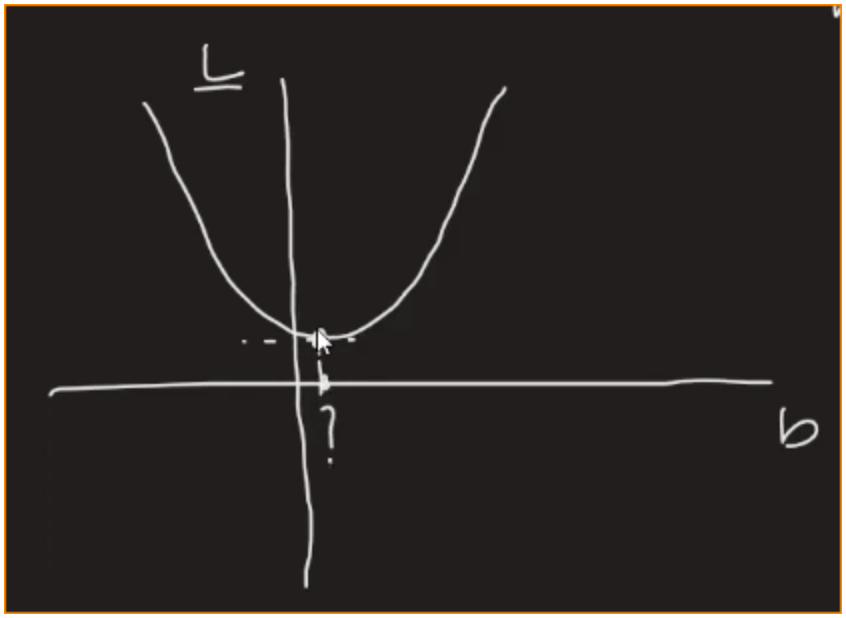
- lets assume correct value of m is already known
- so the equation will be:



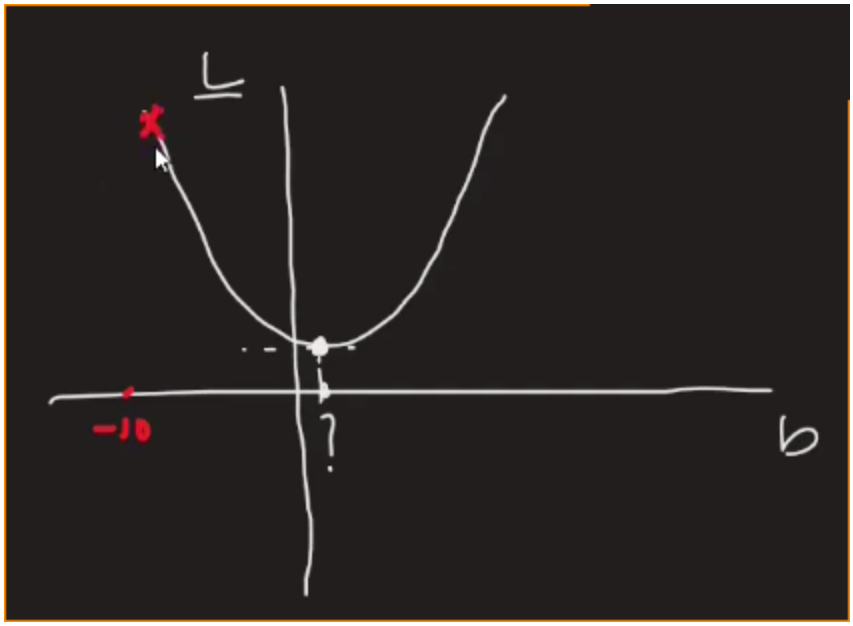
- Now L is the function of b means best fit line can go up or down only since m is constant
- since relationship b/w L & b is square so the graph will be parabolic



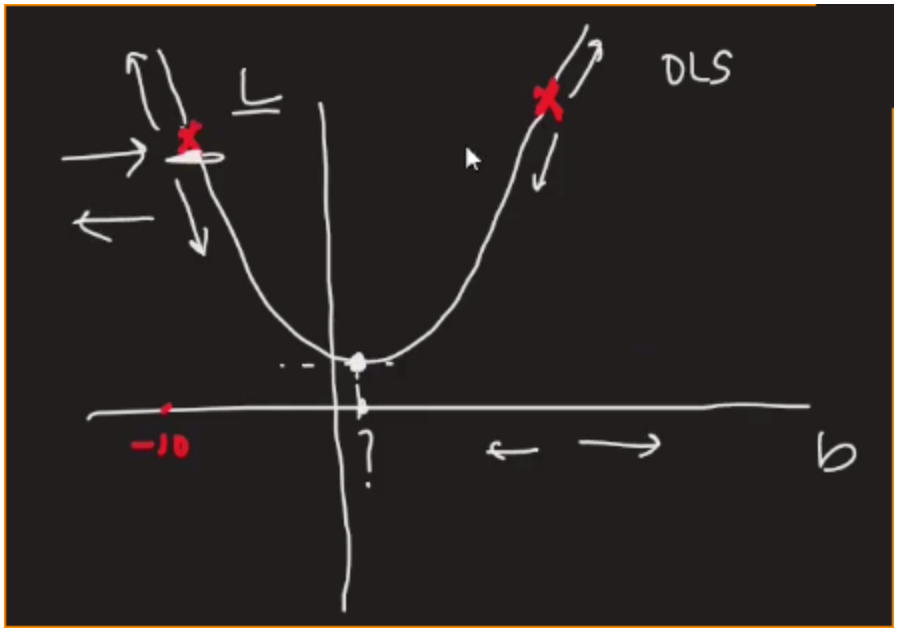
- In this graph.. we have to find such value of b for which L is minimum



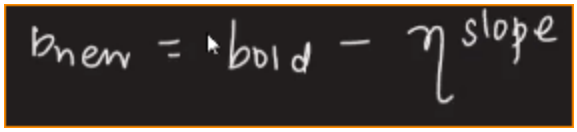
- we could use OLS but it will have challenges when data is huge so we'll use Gradient Descent
- Step1: Select a random b = -10 & the corresponsing loss function



- we need to change the value of b to arrive at the minima point
- so how do we know whether we need to increase or decrease the value of b?

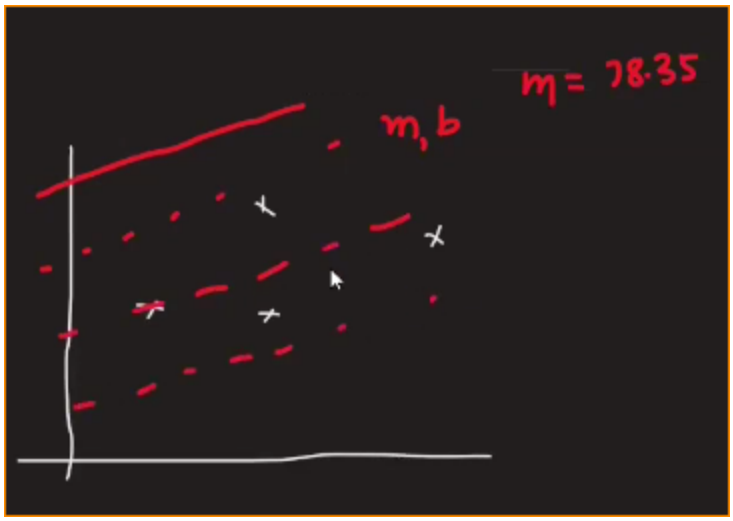


- answer lies withn slope, so we'll find the slope at whatever point we are..
- if slope is +5 then we'll move in negative direction means we need to decrease the value of b
- if slope is -5 then we'll move in positive direction means we need to increase the value of b
- (b_new = b_old - slope) --> There will be a point where b_old & slope will be equal & thats where the slope == 0 & thats our minima
- we also add a tuner called "Learning Rate" (ɳ) to slow down the speed of moving.. else it will take big jumps & might take a lot of time to converge

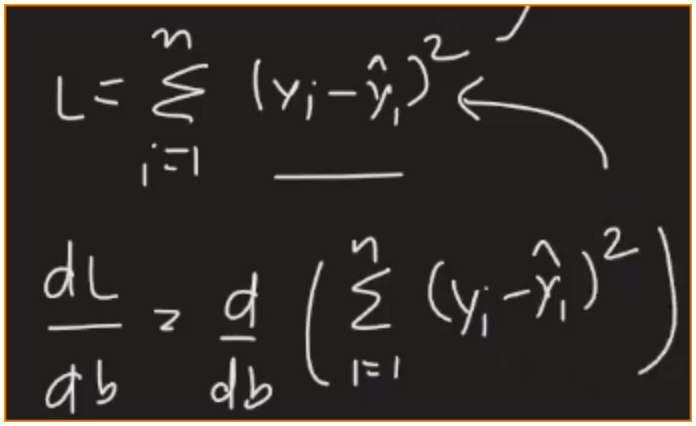


- lets say we initialize it with b value -10 & slope there is calculated as -50 == 40
- so we moved towards the positive side, now calculate slope again, it comes out to be -50
- so we moved to negative direction... thats why we use learning rate to not jump either direction rather move slowly
- keep learning rate value as very less == 0.01 so it will become (-10 - (0.01 * -50)) == -10 + 0.5 == -9.5
- thats what the defintion means: **The idea is to take repetetive steps in the opposite direction of the gradient**
- **But, how would I know where to stop?** : Once bnew - bold is << 0.0001 means it became a very small number so it will not make any effect when we move again
- That means we have actually converged to the solution, **Around the minima** since its a approximation technique
- Or you can run a loop to set number of times.. lets say 1000 & you think in 1000 iterations you'll converge, These iterations are called **EPOCHS**

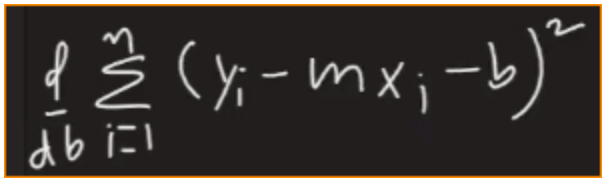
Mathematical Formulation

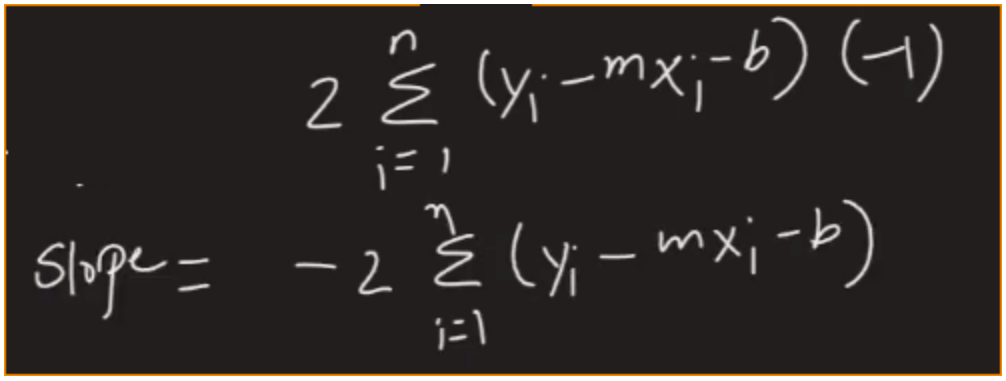


- Lets say we have a graph where we need to find the best fit line
- Also lets assume slope = 78.35 which means we can only move up down, optimize in the terms of b
- Step1: Start with random balue of b
- we need to run loop & update value of **b: bnew = bold - 0.01*slope**
- we need to find slope at b = 0 using diffrentiaion of loss function

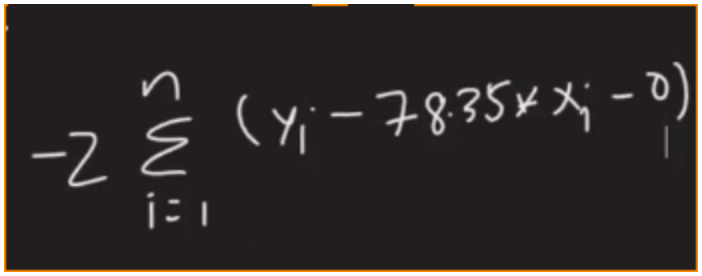


- also replace the value of Ŷ

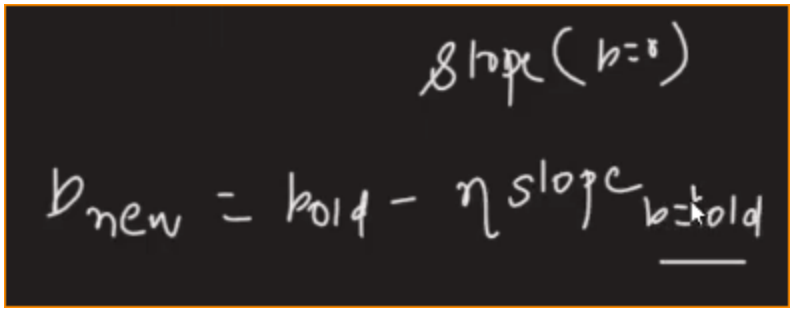




- now replace the value of b = 0 & m = 78.35



- thats when we get the value of slope at b = 0



- we can calculate bnew in loop

Python implimentation

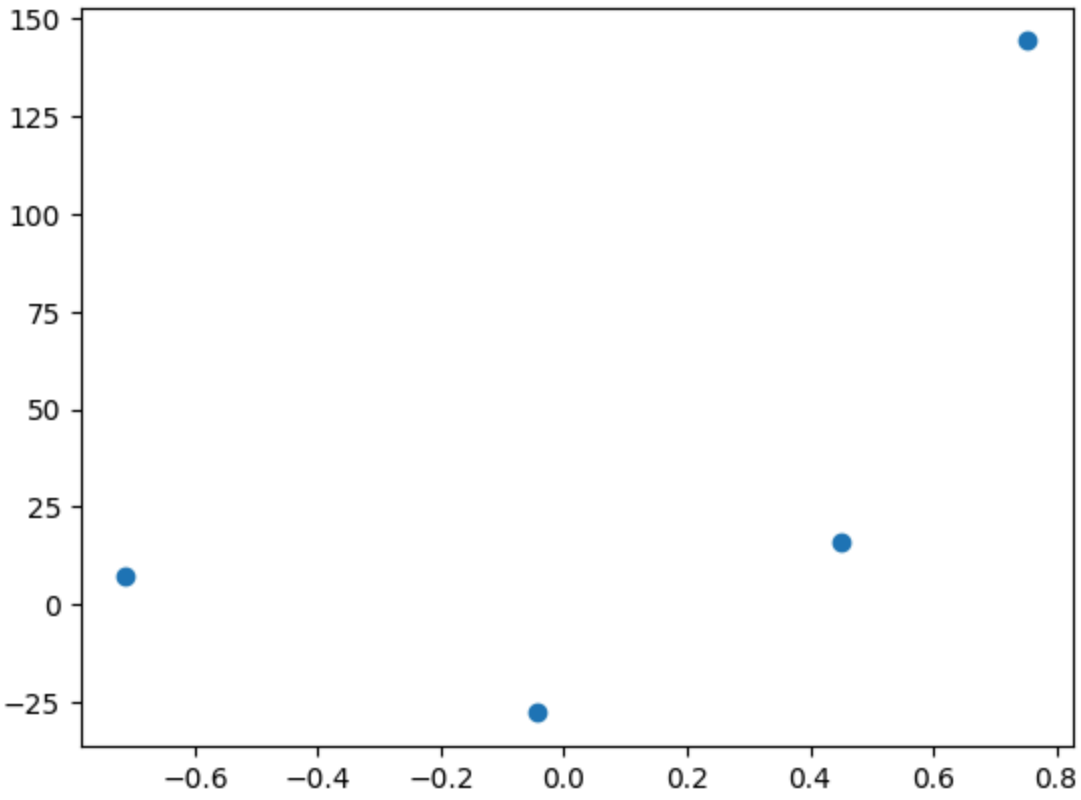
In [1]:
```
#lets first apply ols method & extract slope & find the correct best fit line
#later apply gradient descent & new best fit line should come near to correct line fr
```

In [2]:
```
from sklearn.datasets import make_regression
import numpy as np
```

In [3]:
```
X,y = make_regression(n_samples=4, n_features = 1, n_informative=1, n_targets=1, nois
```

In [4]:
```
import matplotlib.pyplot as plt
plt.scatter(X,y)
```

Out[4]:
```
<matplotlib.collections.PathCollection at 0x1702ee3b9e0>
```



In [5]:
```
#lets apply OLS
from sklearn.linear_model import LinearRegression
```

In [6]:
```
reg = LinearRegression()
reg.fit(X,y)
```

Out[6]:
```
LinearRegression()
```

In [7]:
```
reg.coef_
```

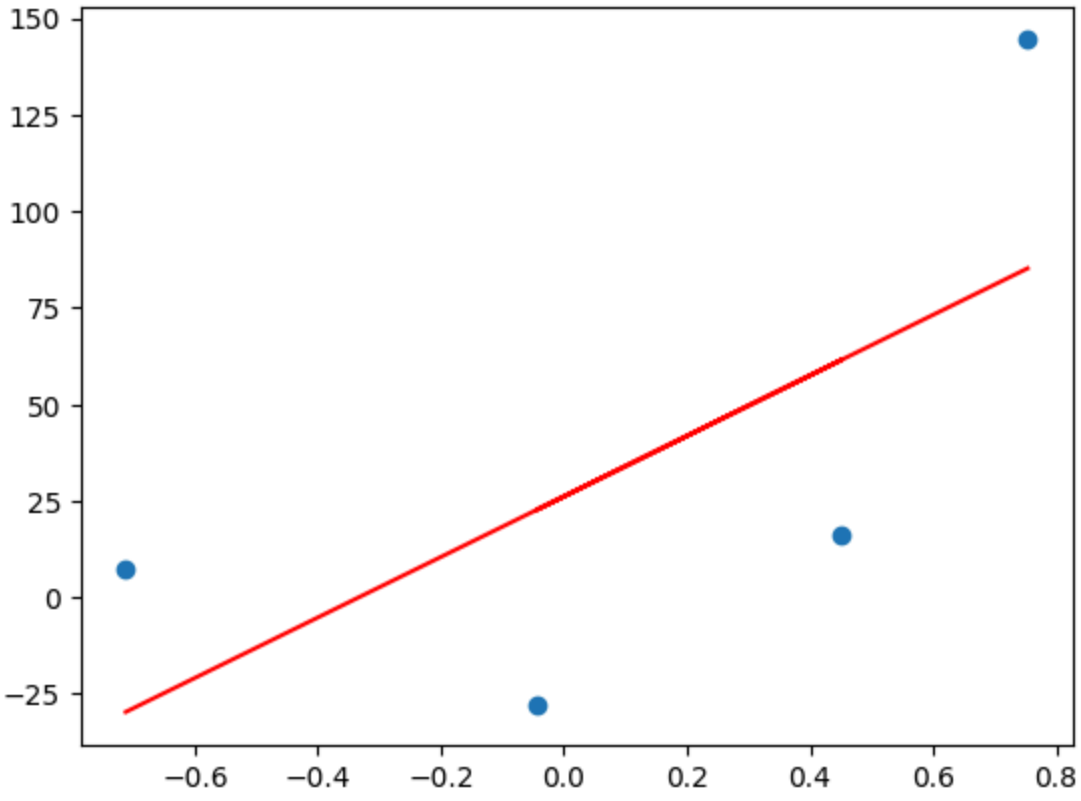
Out[7]:
```
array([78.35063668])
```

In [8]:
```
reg.intercept_
```

Out[8]:
```
26.15963284313262
```

In [9]:
```
plt.scatter(X,y)
plt.plot(X,reg.predict(X), color='red')
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x17031197800>]
```

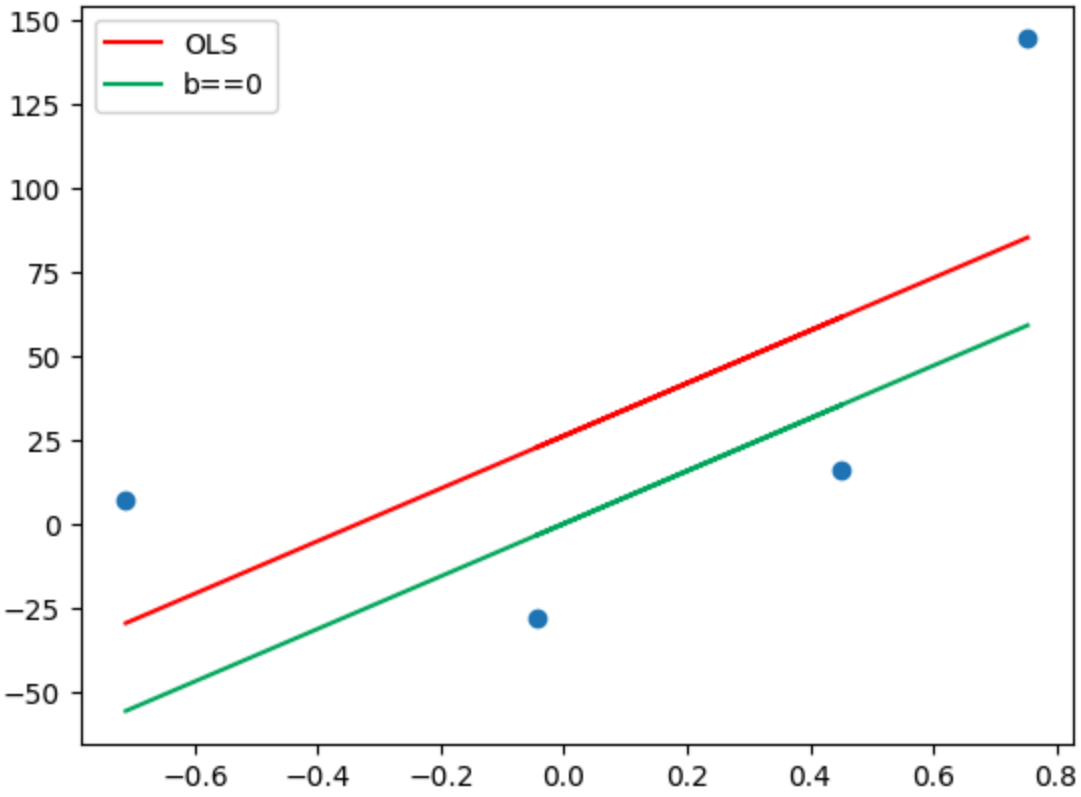


In [10]:
```
#lets apply gradient descent assuming slope = 78.35
#also, starting value for intercept b = 0
y_pred = ((78.35 * X) + 0).reshape(4)
```

In [11]:
```
y_pred
```

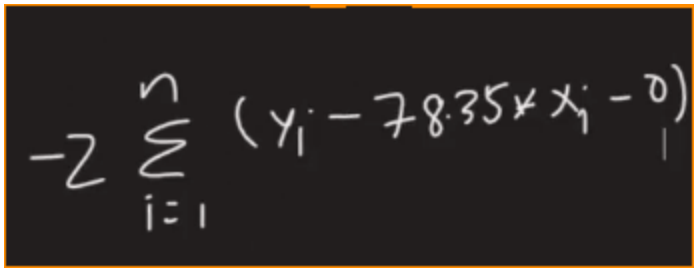
Out[11]: array([-55.81580837, 35.39949674, -3.48681619, 59.05759577])

In [12]:
```
plt.scatter(X,y)
plt.plot(X,reg.predict(X), color='red', label='OLS')
plt.plot(X,y_pred, color='#00a65a', label='b==0')
plt.legend()
plt.show()
```



In [13]:
```
m = 78.35
b = 0
```

- lets put in the value in below formula



In [14]:
```
loss_slope = -2 * np.sum(y-m*X.ravel() - b)
print(f"Slope at b == 0: {loss_slope}")
```

Slope at b == 0: -209.27763408209216

In [15]:
```
#lets take learning rate as 0.1
lr = 0.1
```

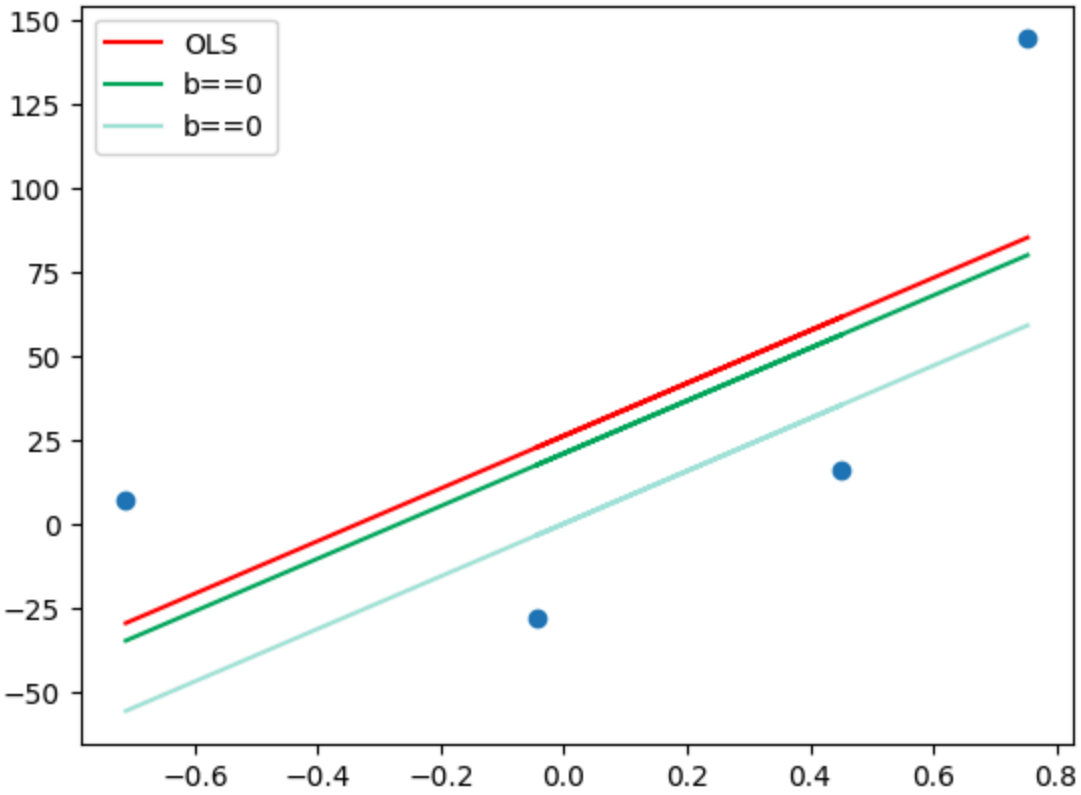
```
step_size = loss_slope*lr
step_size
```

Out[15]: -20.927763408209216

In [16]:
```
#calculating b_new
bnew = b - step_size
print(f"bnew: ",bnew)
```

bnew: 20.927763408209216

In [18]:
```
y_pred1 = ((78.35 * X) + bnew).reshape(4)

plt.scatter(X,y)
plt.plot(X,reg.predict(X), color='red', label='OLS')
plt.plot(X,y_pred1, color='#00a65a', label='b==0')
plt.plot(X,y_pred, color='#A3E4D7', label='b==0')
plt.legend()
plt.show()
```



In [20]:
```
#Lighter shade of green was initial best fit line when b = 0
#darker shade is new best fit line
#red is the OLS best fit line
```

In [21]:
```
#again calculating slope with bnew value
loss_slope = -2 * np.sum(y-m*X.ravel() - bnew)
print(f"Slope at b == 0: {loss_slope}")
```

Slope at b == 0: -41.85552681641843

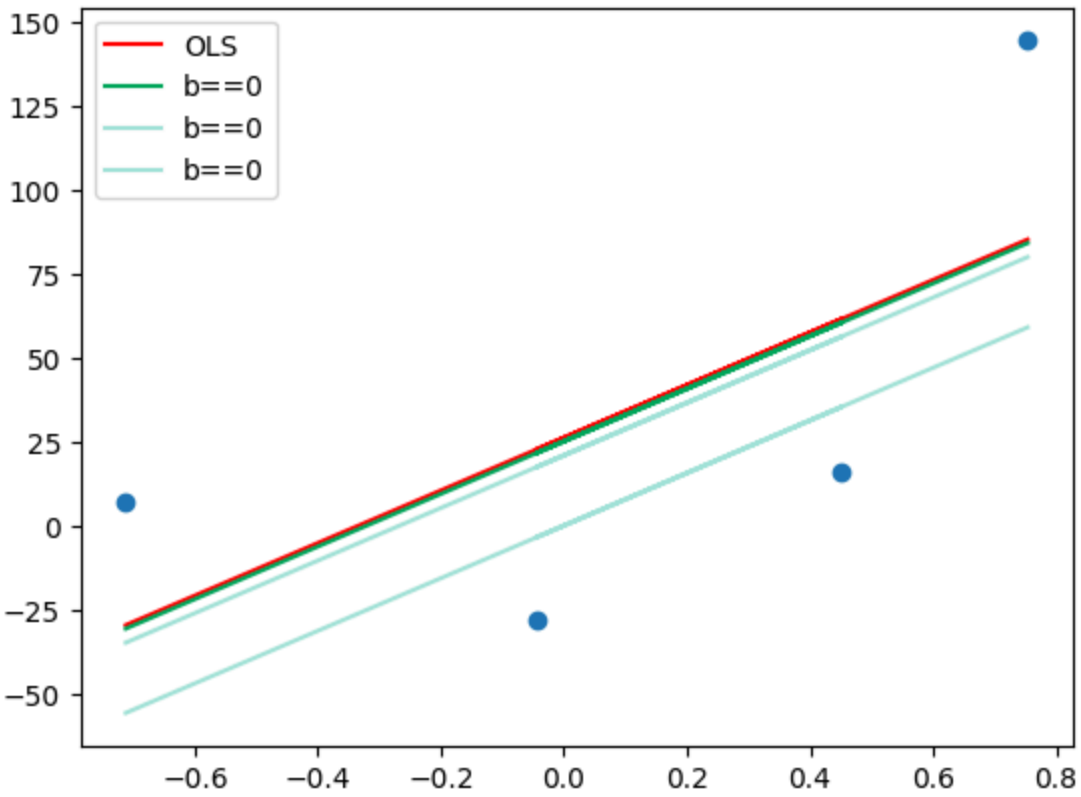
In [22]:
```
#lets take learning rate as 0.1
lr = 0.1

step_size = loss_slope*lr
step_size
```

Out[22]:
```
-4.185552681641844
```

In [24]:
```
bnew = bnew-step_size
bnew
```

Out[24]:
```
25.11331608985106
```

In [26]:
```
y_pred2 = ((78.35 * X) + bnew).reshape(4)

plt.scatter(X,y)
plt.plot(X,reg.predict(X), color='red', label='OLS')
plt.plot(X,y_pred2, color='#00a65a', label='b==0')
plt.plot(X,y_pred1, color='#A3E4D7', label='b==0')
plt.plot(X,y_pred, color='#A3E4D7', label='b==0')
plt.legend()
plt.show()
```



In [27]:
```
#we're almost there, green line is on red line
#lets do 1 more iteration
```

In [28]:
```
#again calculating slope with bnew value
loss_slope = -2 * np.sum(y-m*X.ravel() - bnew)
print(f"Slope at b == 0: {loss_slope}")
```

```
Slope at b == 0: -8.371105363283675
```

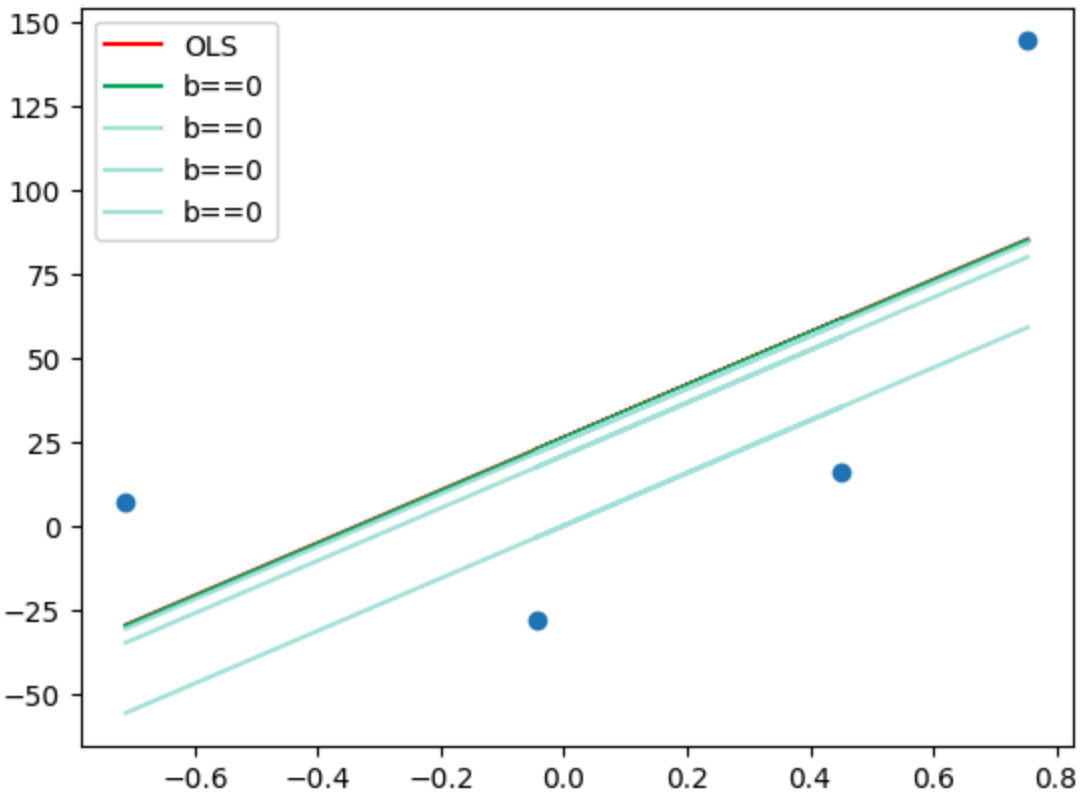
In [29]:
```
#Lets take learning rate as 0.1
lr = 0.1

step_size = loss_slope*lr
step_size
```

Out[29]:
```
-0.8371105363283675
```

In [30]:
```
bnew = bnew-step_size
bnew
```

Out[30]:
```
25.95042662617943
```

In [31]:
```
y_pred3 = ((78.35 * X) + bnew).reshape(4)

plt.scatter(X,y)
plt.plot(X,reg.predict(X), color='red', label='OLS')
plt.plot(X,y_pred3, color='#00a65a', label='b==0')
plt.plot(X,y_pred2, color='#A3E4D7', label='b==0')
plt.plot(X,y_pred1, color='#A3E4D7', label='b==0')
plt.plot(X,y_pred, color='#A3E4D7', label='b==0')
plt.legend()
plt.show()
```



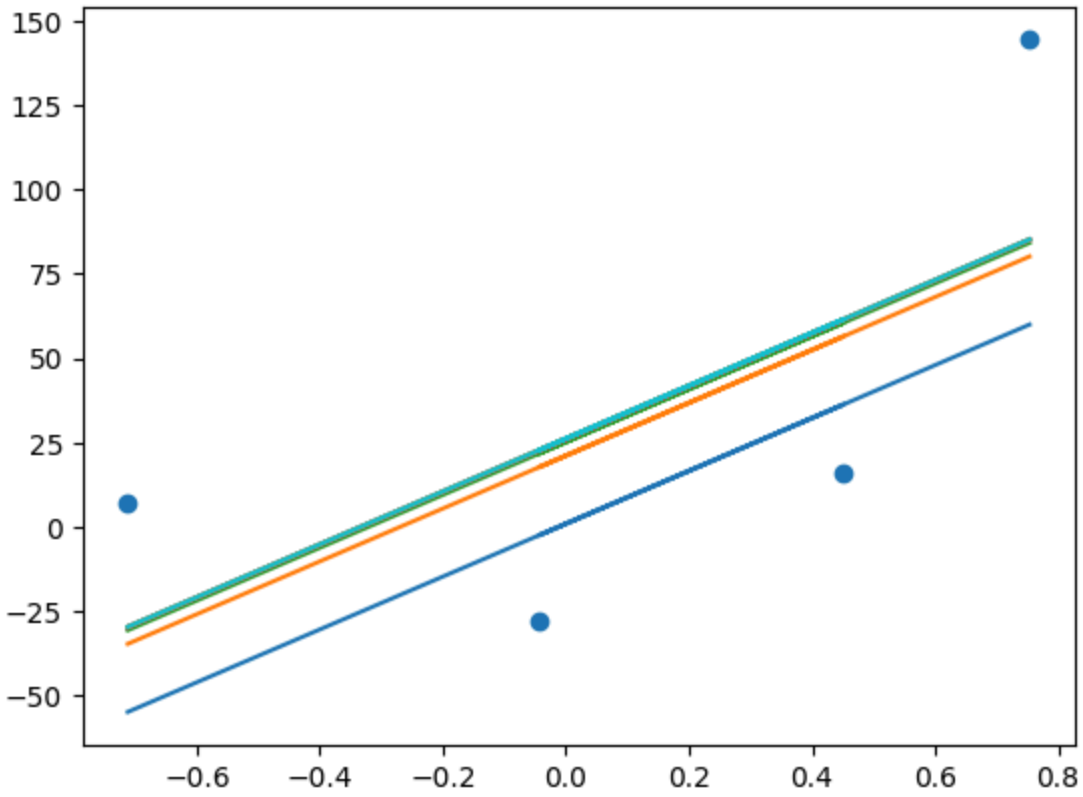
- now we have already converged to the correct answer
- **Beauty of this algorithm is that even if we start with wrong value of b, It will converge to the correct answer**

- when we are away from the correct answer, it takes bigger steps & once we're close it takes small steps, all bcoz of learning rate

Running in loop

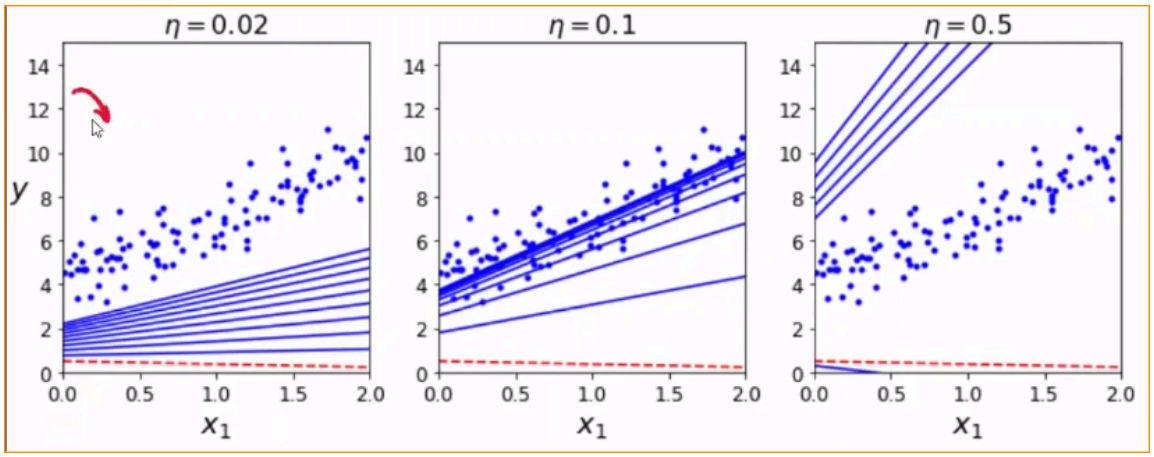
In [44]:
```
b = -100
m = 78.35
lr = 0.1

epochs = 10

for i in range(epochs):
    loss_slope = -2 * np.sum(y - m*X.ravel() - b)
    b = b - (lr * loss_slope)

    y_pred = m * X + b
    plt.plot(X, y_pred)

plt.scatter(X,y)
```

Out[44]:
```
<matplotlib.collections.PathCollection at 0x170311c80e0>
```



In [45]:
```
#Whenever you see its moving here & there & not converging to correct answer
#decrease lr & increase epochs
```

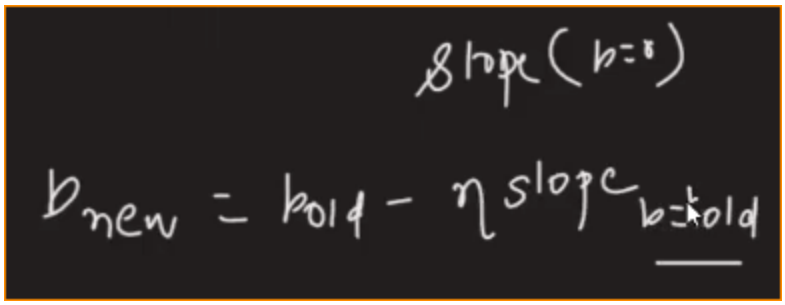
Impact of Learning Rate (Its Hyper Parameter)



- when learning rate is too small, n = 0.02, then we couldnt reach the correct answer
- when learning rate is 0.1 which is optimum, then we were able to converge
- when learning rate is 0.5 which is very high then we couldnt converge & move away
- It shoouldnt be very less or very high, if we keep it very less then it requires more EPOCHS, if its very high then it will never reach the solution

Unversatality of Gradient Descent

- It works on other algorithms not just Linear Regression
- In GD we calculate the value of b using loss function which can be different in Linear regression, Logistic regression or Deep Learning
- we can use those to calculate the slope & converge to the correct soltion as long as lss function is diffrentiable in order to calculate slope
- we need to use below equation which is independent of ml algorithms



Lets calculate using both m & b: The Actual Gradient Descent

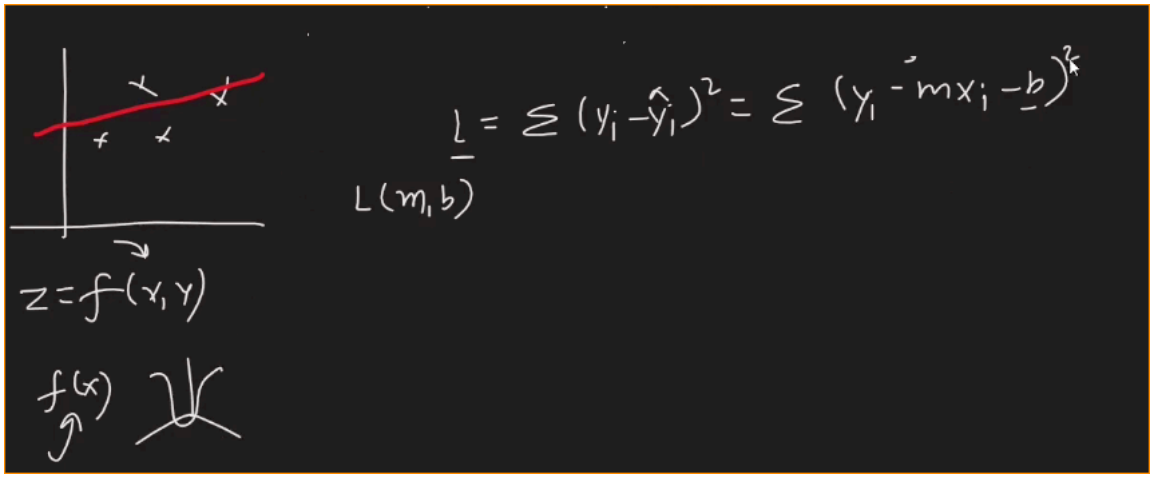
- Above is calculated when m is constant

Steps

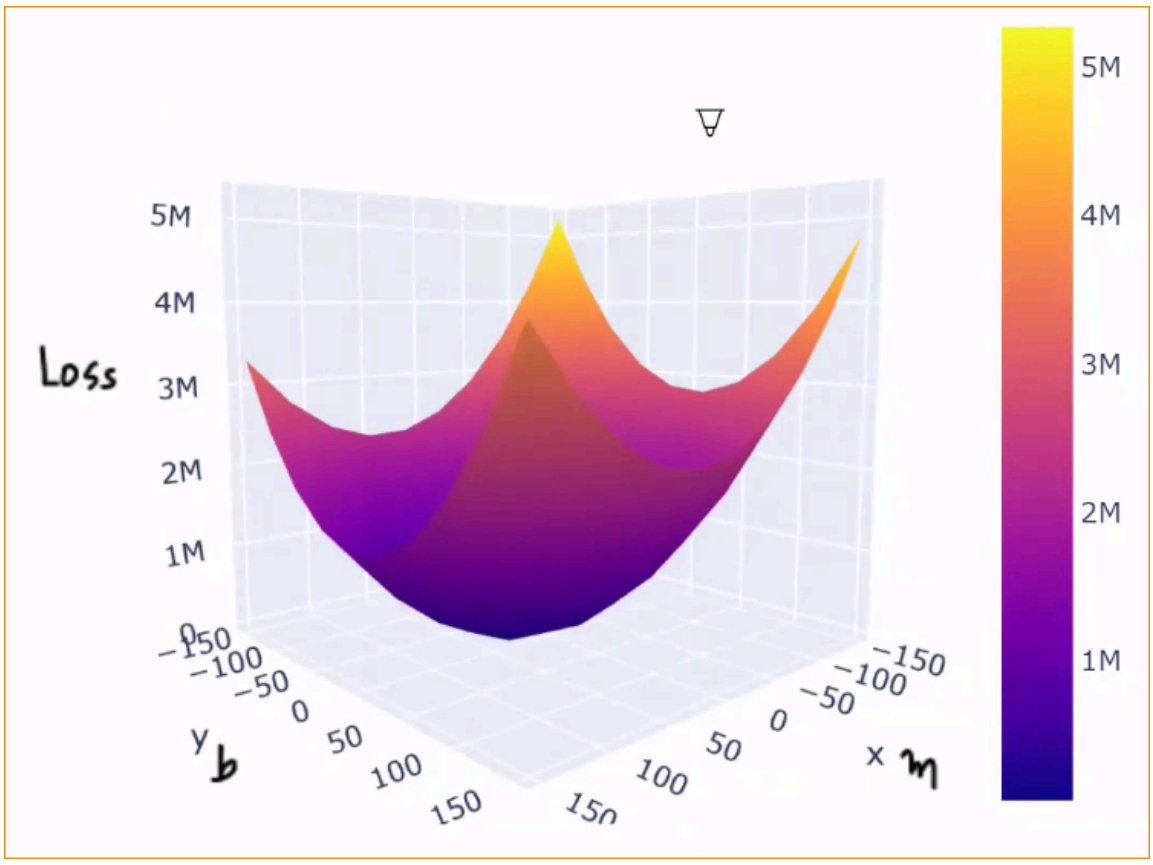
1. Initialize random value of m & b
2. epochs = 100 & lr = 0.01

3. run a loop

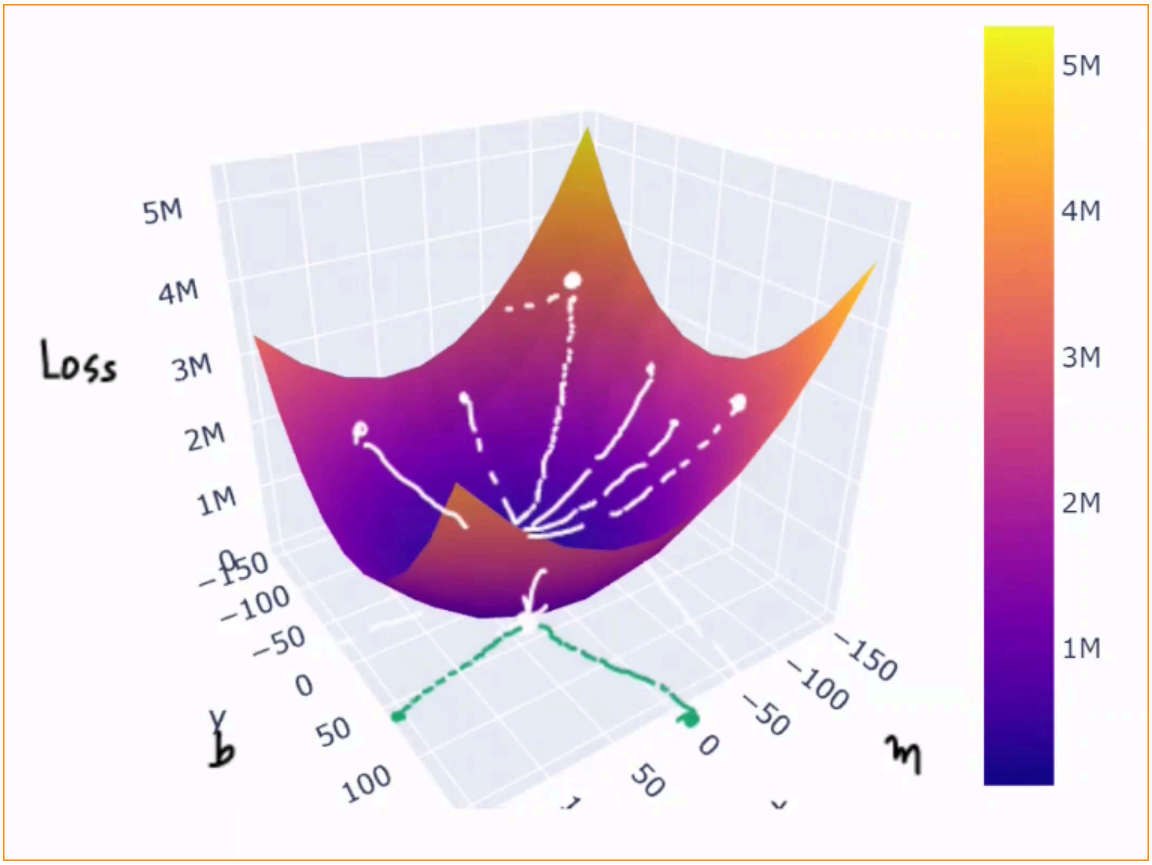
   for in epochs: b = b - n * slope m = m - n * slope



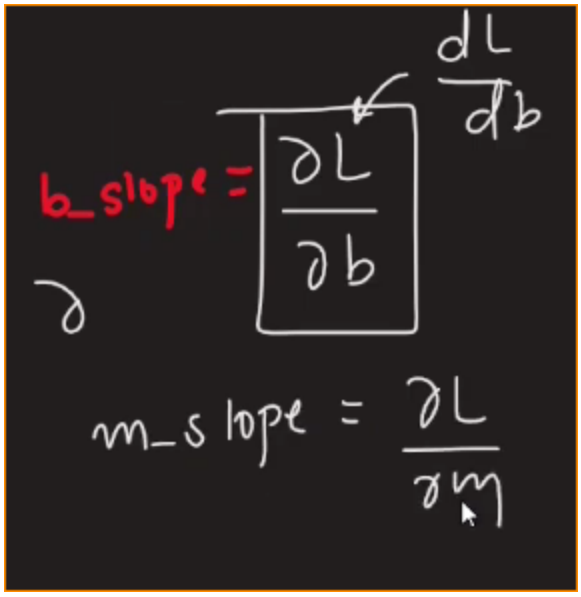
- earlier since m was constant so L was a function of b
- now since its not constant, its a multivariable function L(m,b), Its a 3d parabola function



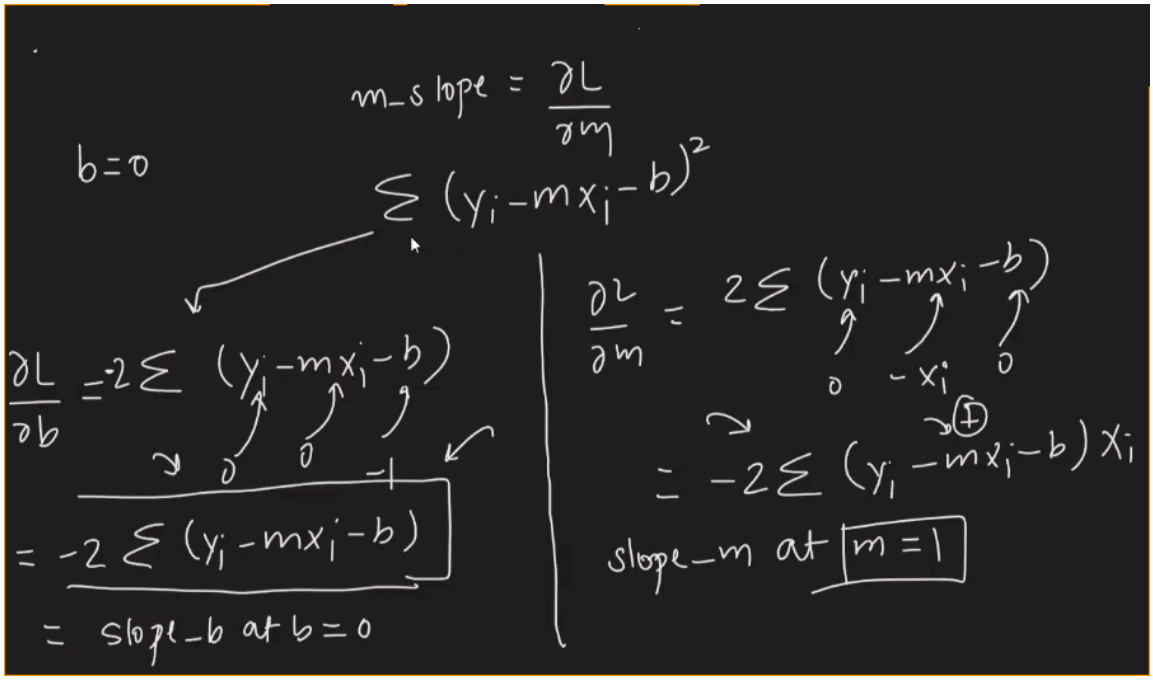
- we need to converge to the point where loss function is minimum & calculate the value of m & b on that point



- At any point in the graph, when we calculate the slope, It will have 2 components
- 1 in the direction of m & 2 in the direction of b
- so we need to perform diffrentiation for both
- diffrentiation of m is direction of m & diffrentiation of b is direction of b
- so the direction is now combination of 2 different variable thats why its called Gradient Descent
- calculate slope in the respect of b & m using partial diffrentiation

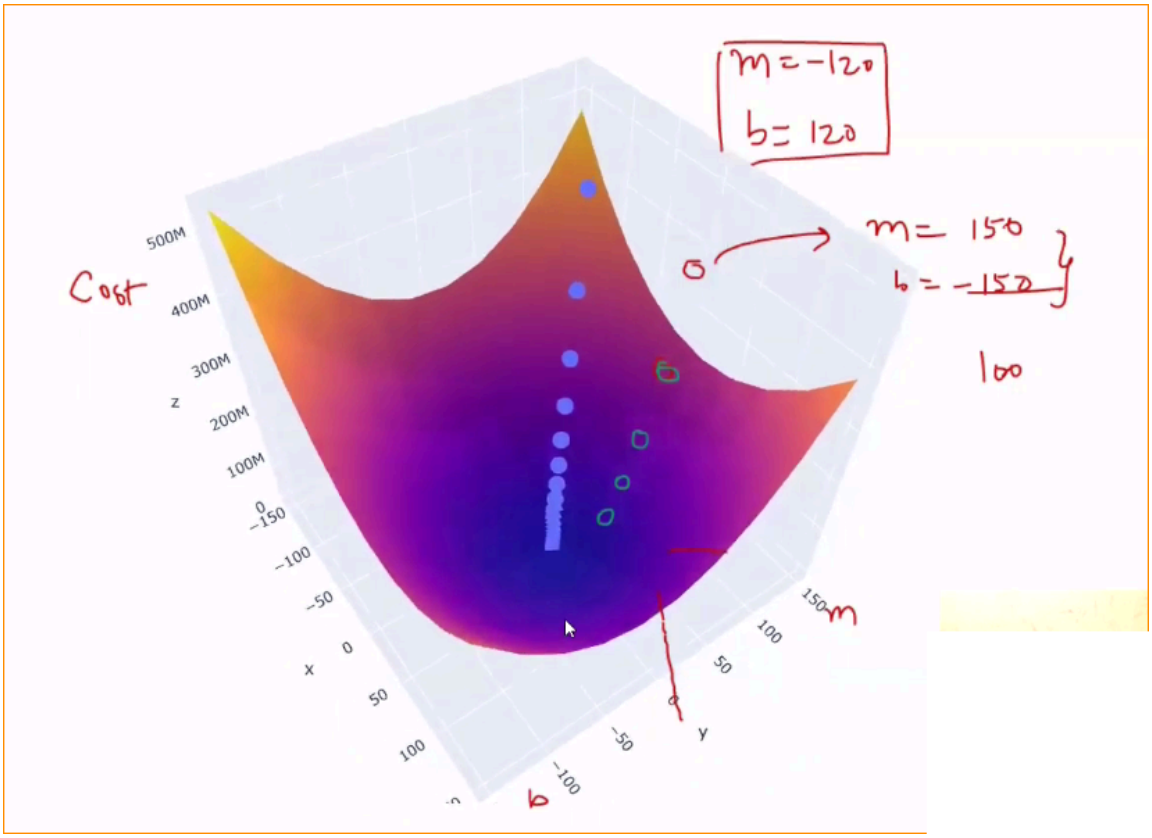


- replace the value of m & b at whereever point you are in



- we need to find slope with respect to m & b
- thier combination will guide where is slope

How Gradient Descent Works?

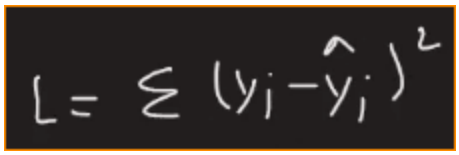


Effect of Learning Rate

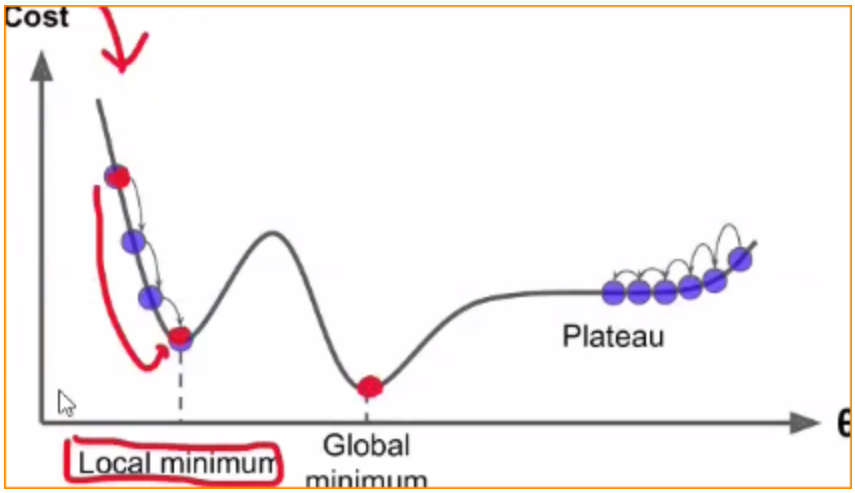
In [47]:
```
#google tool to understand effect of learning rate
#https://developers.google.com/machine-learning/crash-course/fitter/graph
```

Effect of Loss function

- we've use MSE as loss function which is a convex function
- which means if we draw a line b/w 2 points, it will never cross the function
- convex function will only have 1 minima which is global minima
- but in non-convex function, it will have multiple minima



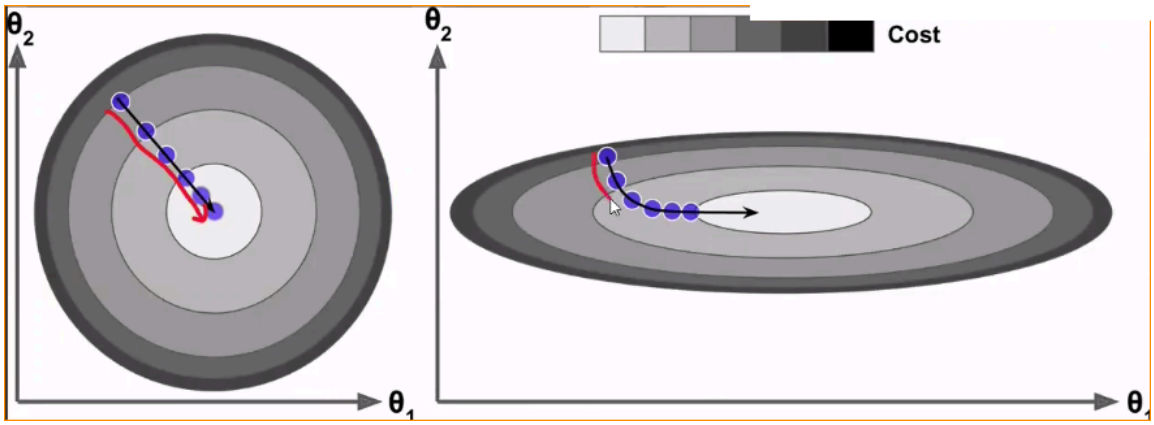
- If your loss function has multiple local minima then its a problem
- it may never reach the best solution & stcuk at local minima whereas global minima is also available



- we can use different initializrion techniques but all this is under depp learning
- There is a also a problem called **Saddle Point** where its flat which requires more EPOCHS hence time complexity will increase

Effect of Data

- When all cols of data are in same scale then it will converge quickly else it will take more time
- Thats why we need to scale the data before applying linear regression



- ***Gradient Descent is one such Algoruthm which teaches us about life, that even if we start wrong aslong as our steps are correct eventually we'll converge to the correct position.***

In [ ]: