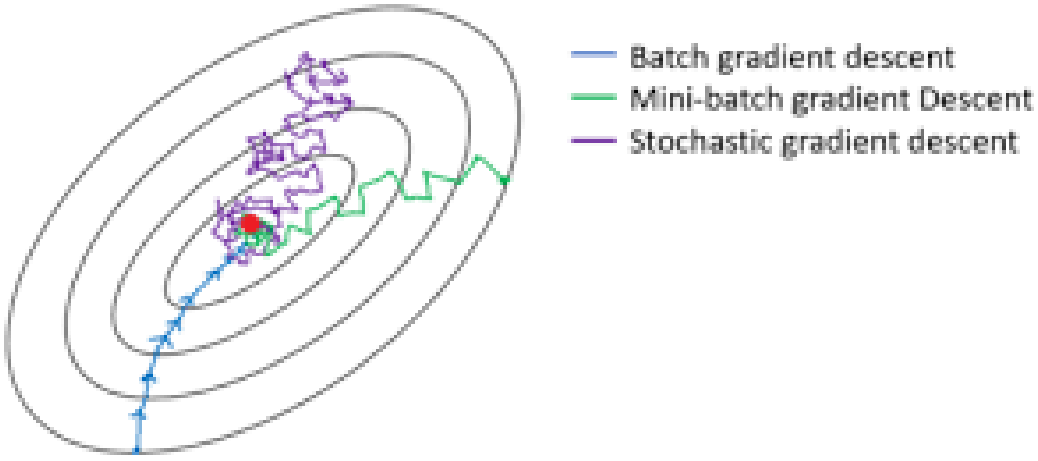
ARIEL UNIVERSITY

Assignment 1 Machine learning

Name - Amit Fallach
ID - 318510070
Linkedin - https://www.linkedin.com/in/amitfallach



For this task, we were required to implement three types of machine learning algorithms: Gradient Descent, Stochastic Gradient Descent (SGD), and mini-batch Gradient Descent (MBGD).

Prior to commencing the assignment, I elected to undertake an in-depth study of the mathematical underpinnings of each algorithm.

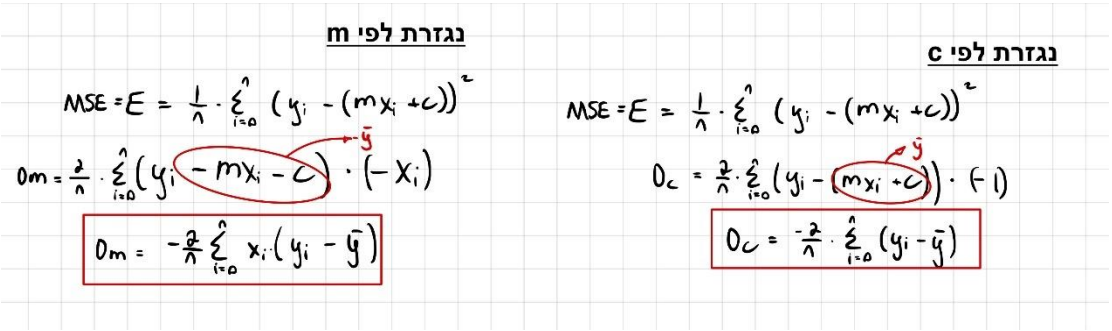
As such, for each Gradient, I will provide a mathematical explanation followed by a description of how the mathematics is translated into code.

I tried to make it in depth, sorry if I took it too far (:

Gradient Descent

Gradient Descent is an iterative optimization algorithm used to minimize the error function between the predicted and actual values (mean squared error) of a model. The algorithm achieves this by adjusting the model's parameters in the direction of the negative gradient until it reaches a local minimum.

To comprehend the mathematics behind the Gradient Descent algorithm, I derived the error function by dividing it by the intercept and slope separately. The results of these partial derivatives can be seen in the attached image. Together with the learning rate, these derivatives affect the rate of change of the parameters.



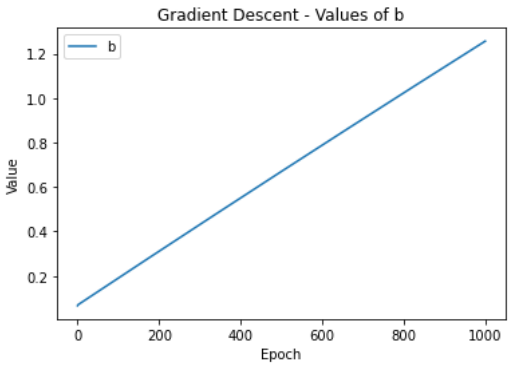
The learning rate (L) determines the step size and the impact of each iteration. A large learning rate can cause the algorithm to skip the minimum because the steps are too significant. A large learning rate is suitable for multivariate functions because it helps to achieve convergence beyond low degrees. On the other hand, a small learning rate can cause the algorithm to converge slowly with small steps, which can require more iterations to reach a high level of accuracy. It is appropriate for functions with a steep slope but not for functions with long valleys, where it can get stuck.

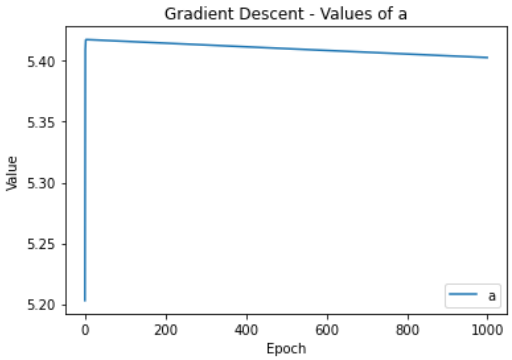
Gradient Descent code

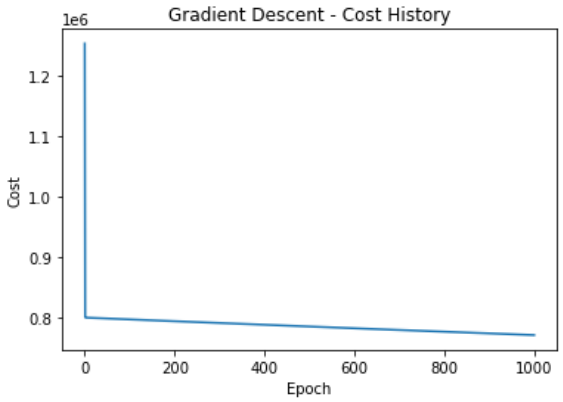
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = pd.read_csv("C:\\Users\\amitf\\Desktop\\amit\\למידת
מכונה\\מטלה 1\\sample.csv")
x,y = data['x'].values,data['y'].values

#calculate the cost function
def costs(a, b, x, y):
    y_pred = a * x + b
    cost = np.sum(np.square(y_pred-y))
    return cost
#-------------------------------------------------------------------
---------
#gradient_descent
a,b,epochs= 0,0,1000
learning_rate = 0.0001 #0
lisa ,lisb , liscost = [],[],[]
def gradient_descent(x, y, a, b, learning_rate, epochs):
    for i in range(epochs):
        y_pred = a * x + b
        error = y - y_pred
#use the partial derivative of the error function and the learning
#rate
        Dm = -(2/len(y)) * np.sum(error * x)
        Dc = -(2/len(y)) * np.sum(error)
        a = a - learning_rate * Dm
        b = b - learning_rate * Dc
        cost = costs(a, b, x, y)
        lisa.append(a) ,lisb.append(b) , liscost.append(cost)
    return a, b, lisa, liscost, lisb

a, b, cost_history, a_history, b_history = gradient_descent(x, y, a,
b, learning_rate, epochs)
print('a: {:.2f}, b: {:.2f}'.format(a, b))


#In order not to be too long, I added the graphs without their code
```

Gradient Descent plots and findings (L = 0.0001)

a: 5.40, b: 1.26



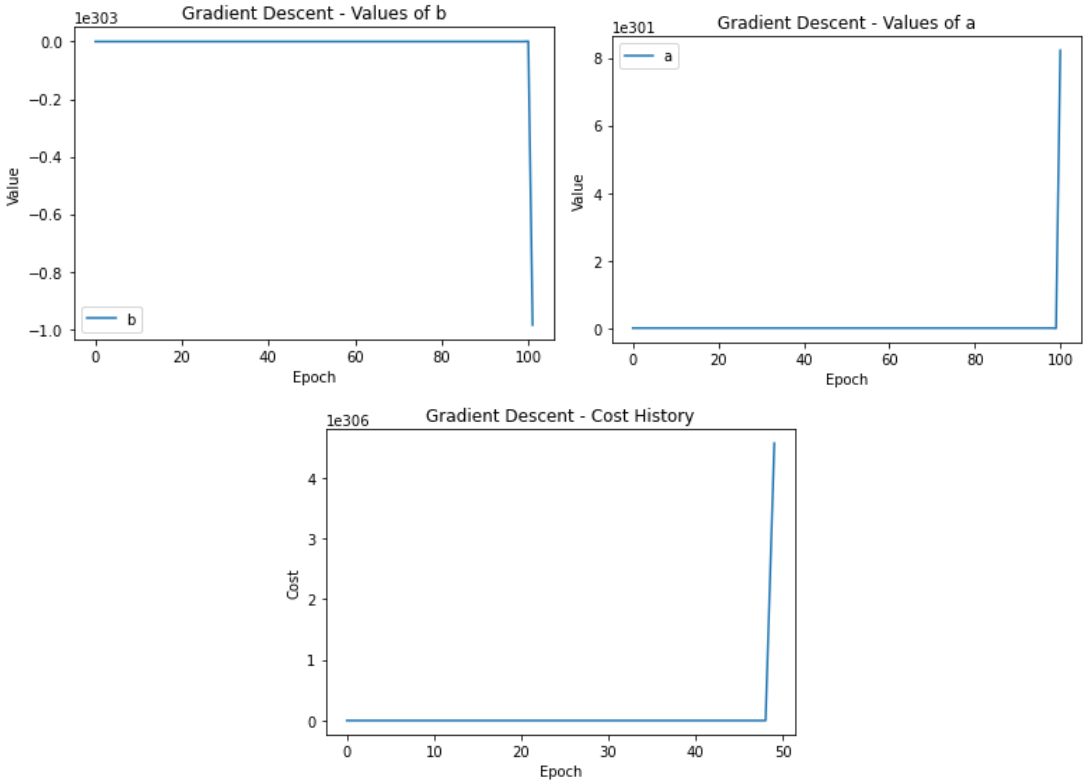




Based on the graphs, it can be inferred that the values of parameter "a" and "cost" remain relatively constant throughout each iteration and it could happened because the gradient reached a minimum extreme point, Therefore, it may be that fewer iterations will be sufficient accordance with the current learning rate . Conversely, the values of intercept "b" exhibit greater precision with each iteration So it may not have reached the optimum point yet or it passed it.

Gradient Descent plots and findings (L = 0. 1)

a: nan, b: nan



Based on the graphs ,we can see that the values of the slope "a" and "cost" reach infinity while the intercept "b" becomes negative infinity after a few iterations. after trying several different learning rates, I have reached the conclusion that an excessively large learning rate results in excessively large parameter updates, leading to divergence of the optimization process. This can cause the model parameters to reach infinite values and fail to converge to an optimal solution.

Stochastic gradient descent - code

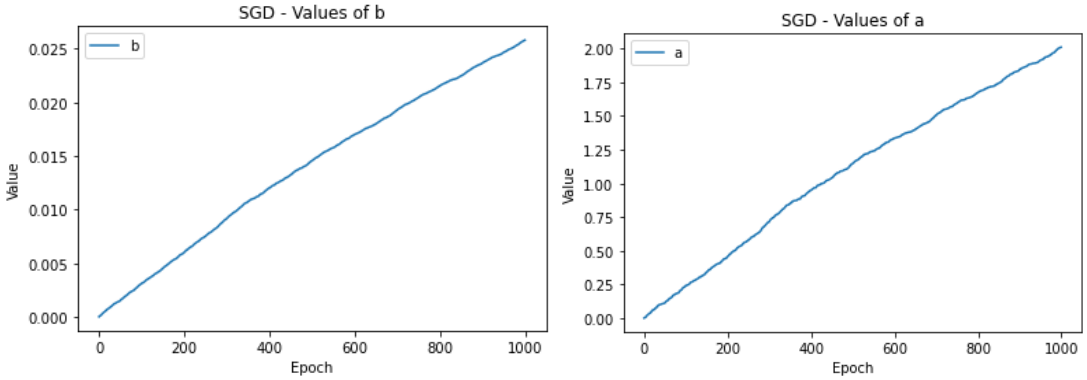
Stochastic Gradient Descent (SGD) is a variant of gradient descent algorithm, where a one random sample is selected to estimate the error . This approach is more computationally efficient and faster than traditional gradient descent but may be less accurate in reaching the optimal solution.

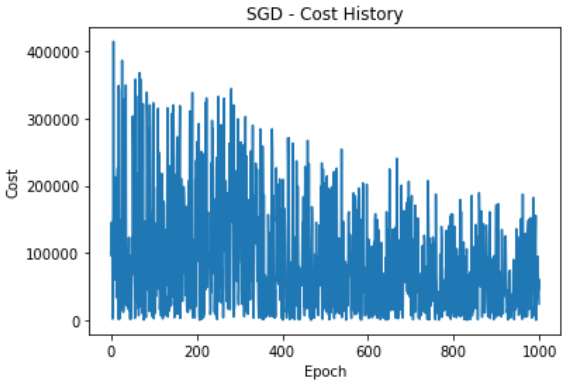
```
#Stochastic gradient descent
a,b,epochs= 0,0,1000
learning_rate = 0.0001 #0.1
lisa ,lisb , liscost = [],[],[]

def SGD(a , b , x, y):
    y_pred = a * x + b
    error = y - y_pred
    Dm = -(2/len(y_shuffled)) * np.sum(error * x)
    Dc = -(2/len(y_shuffled)) * np.sum(error)
    a = a - learning_rate * Dm
    b = b - learning_rate * Dc
    cost = costs(a, b, x, y)
    lisa.append(a)
    lisb.append(b)
    liscost.append(cost)
    return a,b,cost

# loop that mix the data every epoch and take different value
for i in range(epochs):
    idx = np.random.permutation(len(y))
    x_shuffled , y_shuffled= x[idx],y[idx]
    j = np.random.randint(len(y))
    a , b, cost = SGD(a , b , x_shuffled[j], y_shuffled[j])
```

The difference between the stochastic and the normal gradient is a loop that mixes the data every iteration and chooses one sample instead of choosing a vector with all the values.

SGD plots and findings (L = 0.0001)

```
a: 2.00, b: 0.025
```



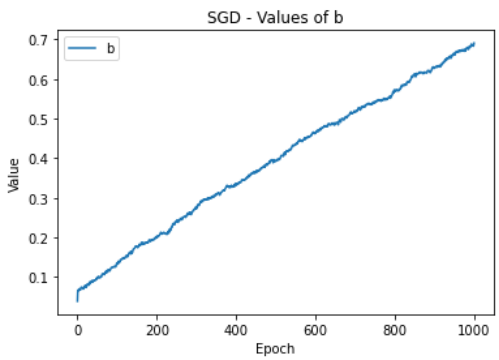


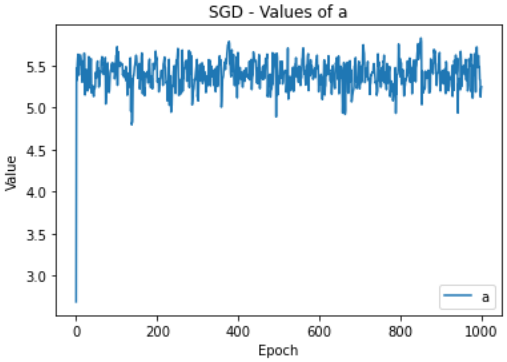
The graphs show an increase in the level of accuracy in each iteration of all values, because I did not see a certain stabilization I concluded that the model did not have enough iterations or that the learning rate was too small to reach an optimum point.

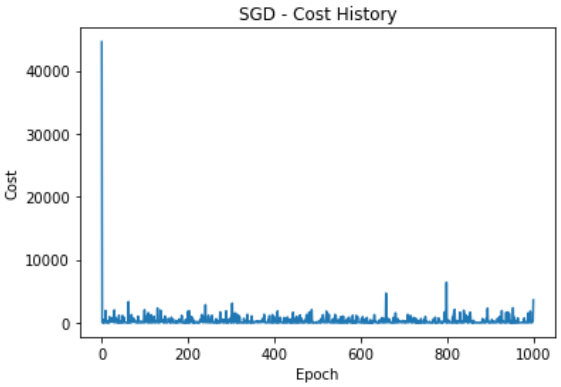
After running a few learning rates, I realized that if the learning rate is too small, like the current example, there are not enough iterations to arrive at a correct answer (with a larger learning rate, the answer is really similar to gradient descent).

SGD plots and findings (L = 0.1)

a : 5.25, b: 0.69







Following to the previous explanation about the SGD, thanks to the stabilization of the slope "a" and the "cost" it can be seen that in large steps the algorithm quickly reaches the optimum point. In addition, you can notice on the graph of "b" that there is still changes and it hasn't stabilized yet, so either it did not reach the optimum point or it missed it.

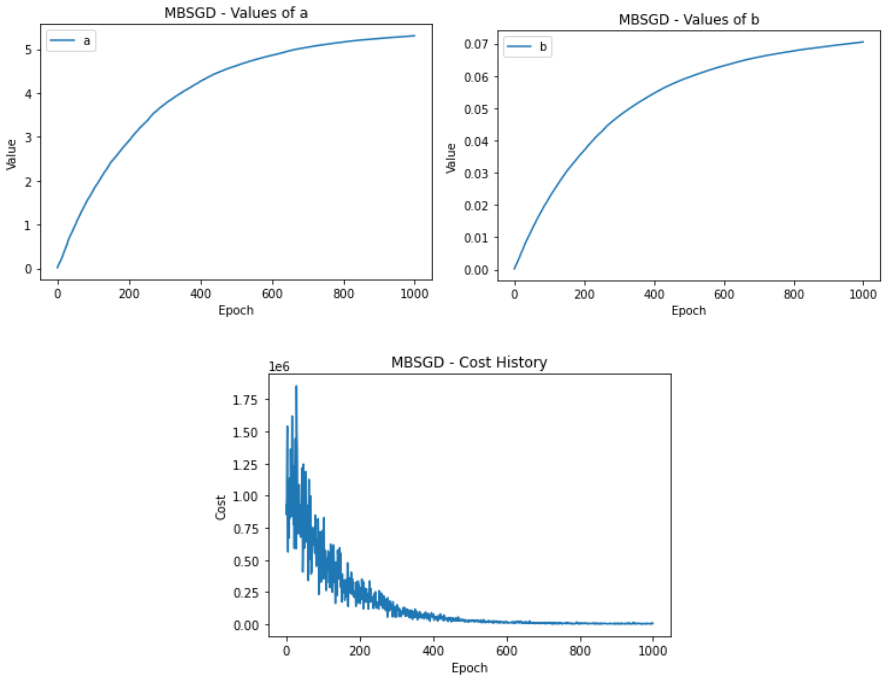
Mini Batch gradient descent – code

The mathematical difference in the code between the MBGD and the SDG is that the loop that calls the function will call a fixed amount of random samples each time.

```
# Mini Batch Stochastic gradient descent
a, b, epochs = 0, 0, 1000
learning_rate = 0.0001  # 0.1
x, y = data['x'].values, data['y'].values
lisa, lisb, liscost = [], [], []


def MBSGD(a, b, x, y):
    y_pred = a * x + b
    error = y - y_pred
    Dm = -(2 / len(y_shuffled)) * np.sum(error * x)
    Dc = -(2 / len(y_shuffled)) * np.sum(error)
    a = a - learning_rate * Dm
    b = b - learning_rate * Dc
    cost = costs(a, b, x, y)
    lisa.append(a)
    lisb.append(b)
    liscost.append(cost)
    return a, b, cost


for i in range(1000):
    idx = np.random.permutation(len(y))
    x_shuffled = x[idx]
    y_shuffled = y[idx]
    j = np.arange(8)
    a, b, cost = MBSGD(a, b, x_shuffled[j], y_shuffled[j])

# Print the learned parameters
print('a: {:.2f}, b: {:.2f}, cost: {:.2f}'.format(a, b, cost))
```

MBGD plots and findings (L = 0.0001)

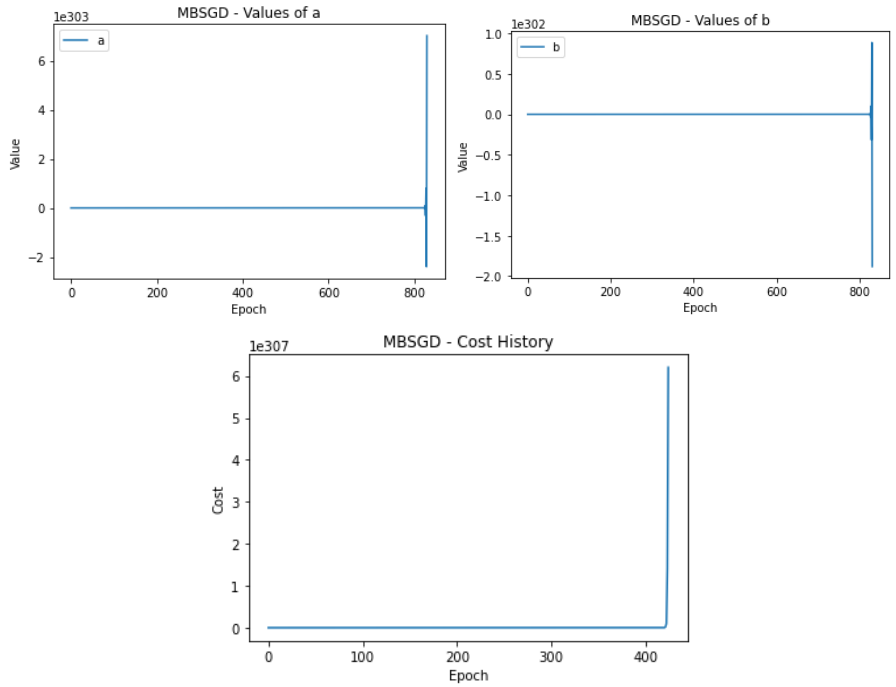
`a: 5.30, b: 0.07`



It can be seen that the parameters "a" and "b" reach a kind of asymptote or a balance point at a certain stage in the model and also "cost" balances around certain values and therefore I chose to conclude that the model has reached an optimum point.

MBGD plots and findings (L = 0. 1)

a: nan, b: nan



Based on the graphs ,we can see that the values of the slope "a" and "b" entertained while the intercept "cost" becomes infinity after a few iterations. after trying several different learning rates, I have reached the conclusion that an excessively large learning rate results in excessively large parameter updates, leading to divergence of the optimization process. This can cause the model parameters to reach infinite values and fail to converge to an optimal solution.