Unit 8

gradient descend cost of function

In this activity, I worked on running a code in Google Colab related to the gradient descent concept covered in this unit. I experimented by adjusting the number of iterations and the learning rate to observe how the cost function decreases over time. Below is the initial code I started with:

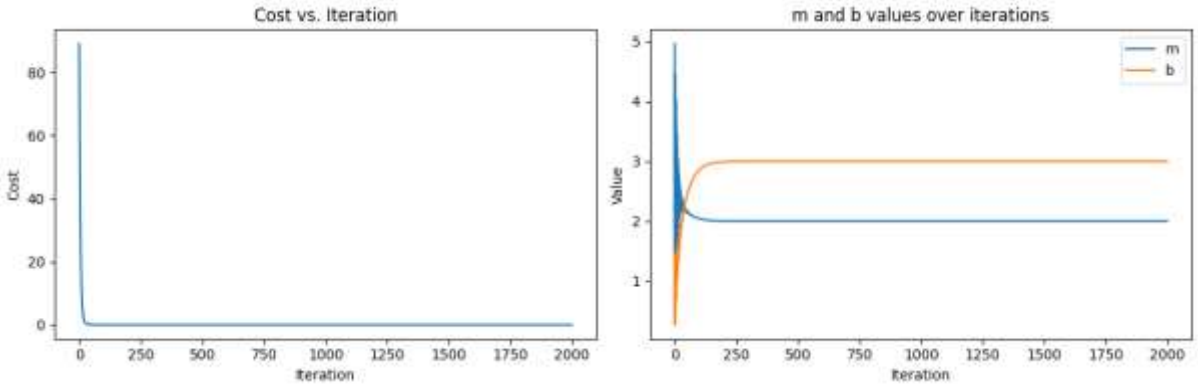
```
import numpy as np

def gradient_descent(x,y):
    m_curr = b_curr = 0
    iterations = 100       #change value
    n = len(x)
    learning_rate = 0.08   #change value

    for i in range(iterations):
        y_predicted = m_curr * x + b_curr
        cost = (1/n) * sum([val**2 for val in (y-y_predicted)])
        md = -(2/n)*sum(x*(y-y_predicted))
        bd = -(2/n)*sum(y-y_predicted)
        m_curr = m_curr - learning_rate * md
        b_curr = b_curr - learning_rate * bd
        print ("m {}, b {}, cost {} iteration {}".format(m_curr,b_curr,cost, i))

x = np.array([1,2,3,4,5])
y = np.array([5,7,9,11,13])

gradient_descent(x,y)
```



Looking at the m and b value plotted against iteration I realised that there is a lot of fluctuation initially which made me realise that the learning rate is to fast.This is what I observed during the activity:

- Initially, the cost was **0.004120600119124239**.
- I increased the number of iterations from 100 to 150, which reduced the cost to **0.0002655301332288879**.

- Next, I increased the learning rate from 0.08 to 0.5. However, this resulted in a warning message: RuntimeWarning: overflow encountered in scalar add. This warning occurs when the computed result exceeds the data type's capacity, indicating that the learning rate was too high.
- To address this, I reduced the learning rate to 0.1. Unfortunately, this gave me a very large cost value of **2.123861426411264e+42**.
- Lowering the learning rate further to 0.05 resulted in a cost of **0.005838643145638964**. Based on these results, I concluded that a learning rate of 0.08 was optimal.

I then experimented with increasing the number of iterations to 2000. I observed that as the number of iterations increased, the cost continued to decrease. However, the mmm and bbb values, along with the cost, stabilized at iteration 1226. This indicated that 1226 iterations were optimal for achieving the lowest cost.

Finally, I decided to plot a graph to visualize the results. Below is the graph showing the behavior with 2000 iterations:

I adjusted the learning rate to **0.04** and set the number of iterations to **2381**, which resulted in the lowest cost function value of **5.616689645173604e-29**.