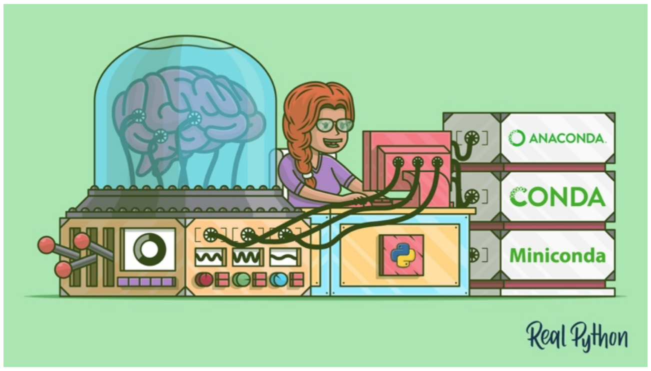
Gradient Descent

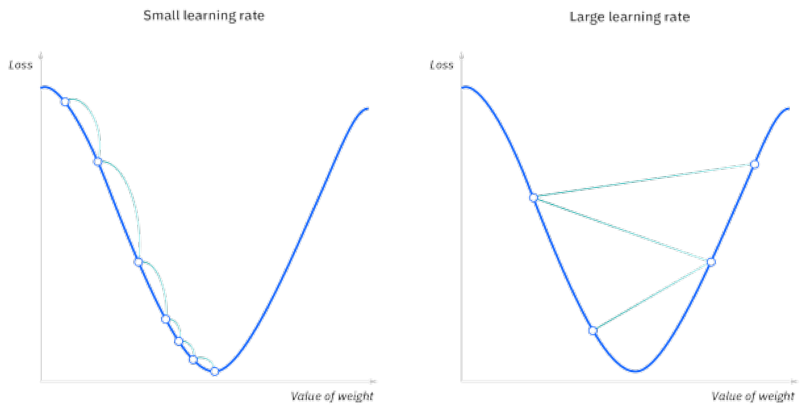
Name: Yogev Ladani
Id:207754524



About Gradient Descent:
Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error. Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.

The possible error : MSE=1/N*∑i=1,n(yi−(mxi+c))2
As a and b are our weights.

Learning rate : is the size of the steps that are taken to reach the minimum. This is typically a small value, and it is evaluated and updated based on the behavior of the cost function.
High learning rates result in larger steps but risks overshooting the minimum. Conversely, a low learning rate has small step sizes



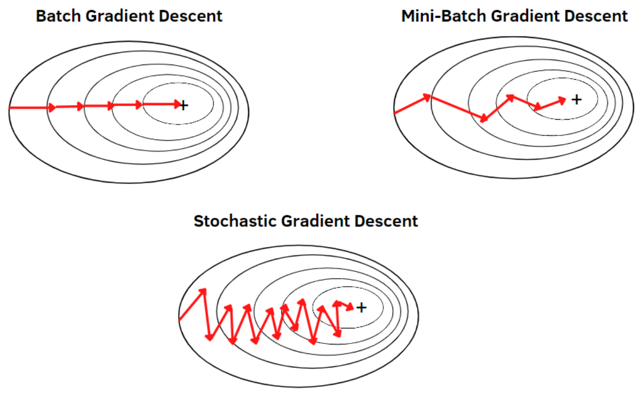
Types of gradient descent

There are three types of gradient descent learning algorithms: batch gradient descent, stochastic gradient descent and mini-batch gradient descent.

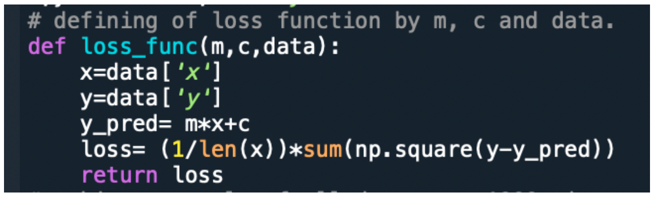
Batch gradient descent- sums the error for each point in a training set, updating the model only after **all** training examples have been evaluated and return this as number of iteration we determine (in our assignment over 1000 times). This process referred to as a training epoch. While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store all of the data into memory. Batch gradient descent also usually produces a stable error gradient and convergence, but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

Stochastic gradient descent (SGD)- The algorithm takes each iteration only one sample from the data set and updates each training example's parameters one at a time. Since you only need to hold one training example, they are easier to store in memory. While these frequent updates can offer more detail and speed, it can result in losses in computational efficiency when compared to batch gradient descent. Its frequent updates can result in noisy gradients, but this can also help find the minimum global.

Mini-batch gradient descent combines concepts from both batch gradient descent and stochastic gradient descent. It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach strikes a balance between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

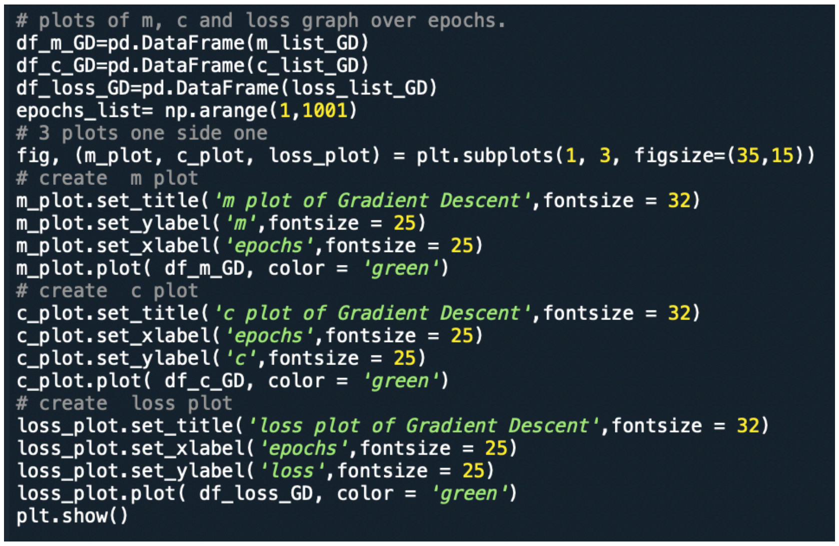


Loss function:



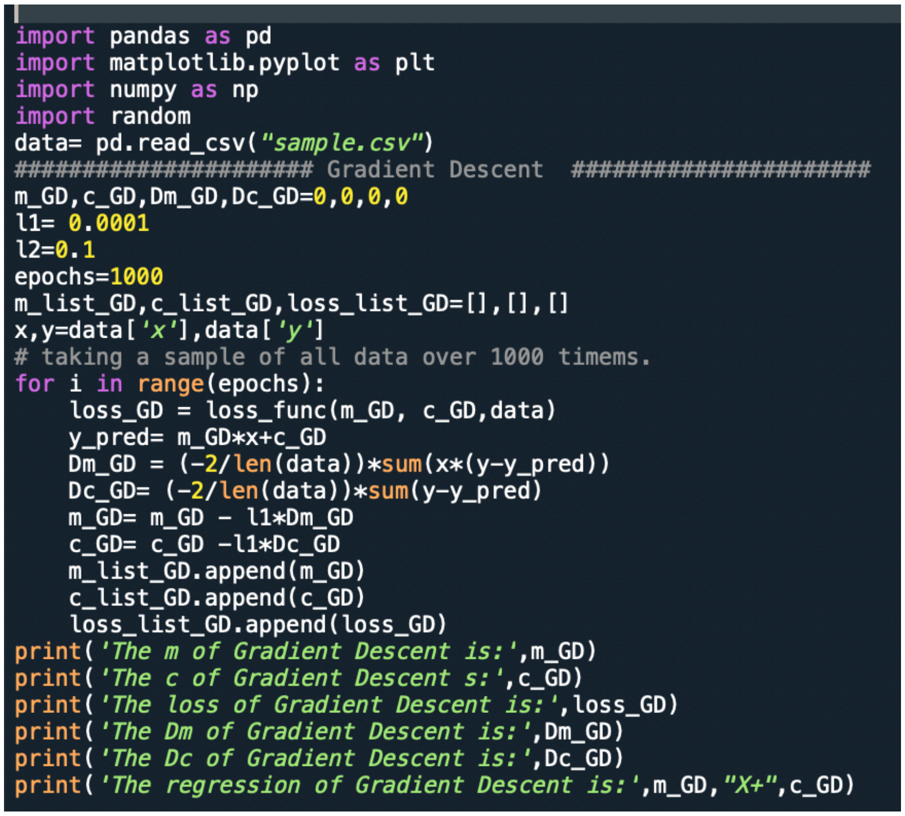
```python
# defining of loss function by m, c and data.
def loss_func(m,c,data):
    x=data['x']
    y=data['y']
    y_pred= m*x+c
    loss= (1/len(x))*sum(np.square(y-y_pred))
    return loss
```

The plotting code:
For each type of Gradient Descent the plot code is the same (just with a different name) so I attach it only once.



```python
# plots of m, c and loss graph over epochs.
df_m_GD=pd.DataFrame(m_list_GD)
df_c_GD=pd.DataFrame(c_list_GD)
df_loss_GD=pd.DataFrame(loss_list_GD)
epochs_list= np.arange(1,1001)
# 3 plots one side one
fig, (m_plot, c_plot, loss_plot) = plt.subplots(1, 3, figsize=(35,15))
# create  m plot
m_plot.set_title('m plot of Gradient Descent',fontsize = 32)
m_plot.set_ylabel('m',fontsize = 25)
m_plot.set_xlabel('epochs',fontsize = 25)
m_plot.plot( df_m_GD, color = 'green')
# create  c plot
c_plot.set_title('c plot of Gradient Descent',fontsize = 32)
c_plot.set_xlabel('epochs',fontsize = 25)
c_plot.set_ylabel('c',fontsize = 25)
c_plot.plot( df_c_GD, color = 'green')
# create  loss plot
loss_plot.set_title('loss plot of Gradient Descent',fontsize = 32)
loss_plot.set_xlabel('epochs',fontsize = 25)
loss_plot.set_ylabel('loss',fontsize = 25)
loss_plot.plot( df_loss_GD, color = 'green')
plt.show()
```

Batch gradient descent code:



```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import random
data= pd.read_csv("sample.csv")
###################### Gradient Descent  ######################
m_GD,c_GD,Dm_GD,Dc_GD=0,0,0,0
l1= 0.0001
l2=0.1
epochs=1000
m_list_GD,c_list_GD,loss_list_GD=[],[],[]
x,y=data['x'],data['y']
# taking a sample of all data over 1000 timems.
for i in range(epochs):
    loss_GD = loss_func(m_GD, c_GD,data)
    y_pred= m_GD*x+c_GD
    Dm_GD = (-2/len(data))*sum(x*(y-y_pred))
    Dc_GD= (-2/len(data))*sum(y-y_pred)
    m_GD= m_GD - l1*Dm_GD
    c_GD= c_GD -l1*Dc_GD
    m_list_GD.append(m_GD)
    c_list_GD.append(c_GD)
    loss_list_GD.append(loss_GD)
print('The m of Gradient Descent is:',m_GD)
print('The c of Gradient Descent s:',c_GD)
print('The loss of Gradient Descent is:',loss_GD)
print('The Dm of Gradient Descent is:',Dm_GD)
print('The Dc of Gradient Descent is:',Dc_GD)
print('The regression of Gradient Descent is:',m_GD,"X+",c_GD)
```

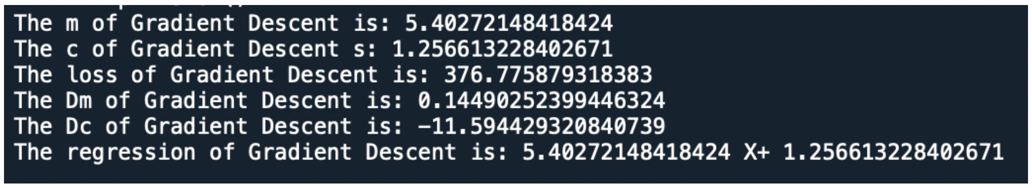
Here we need to define the initial value of m and c parameters and epochs, we also define the learning rate values.
The function over all data as the number of epochs, updates the of m and c and append the value of m, c, and loss into the list for the plots.
Finally, we print the values of all parameters and the regression.

Note- if we want to change the learning rate, we need to replace l1 with l2.

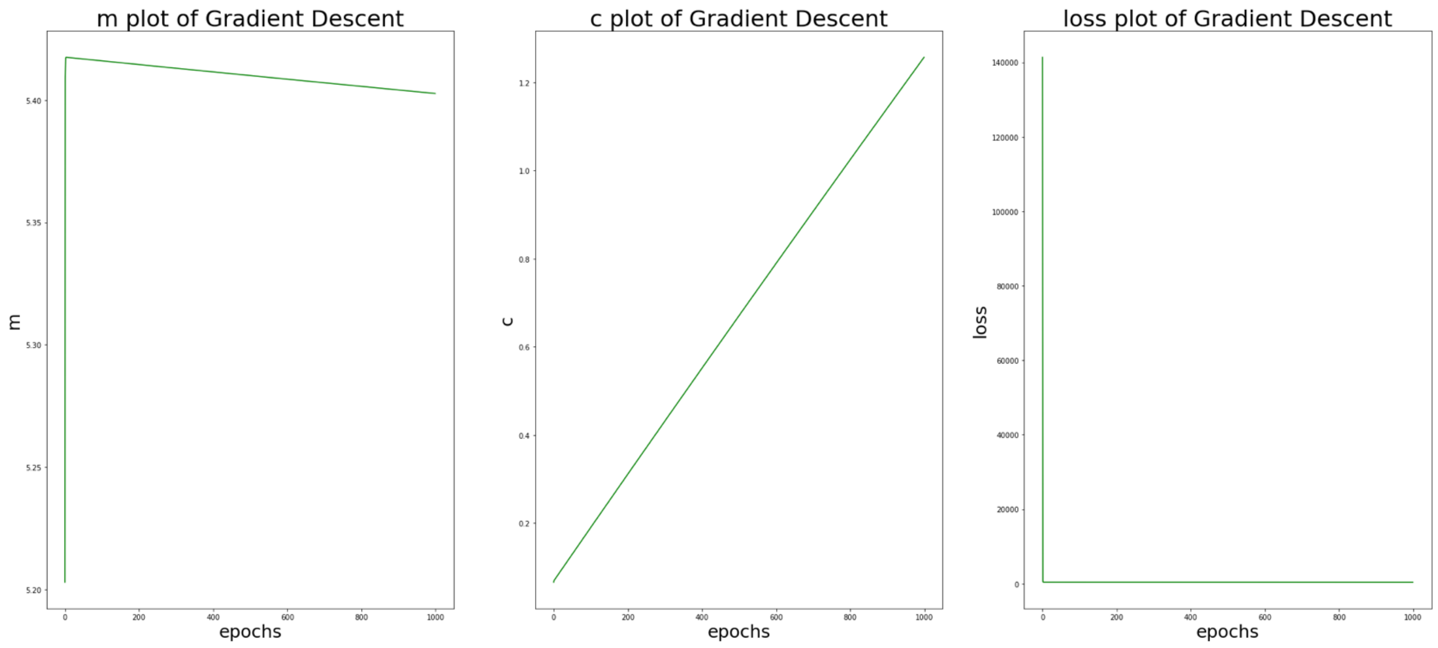
Batch gradient descent graph:

1. Learning rate= 0.0001

Results:



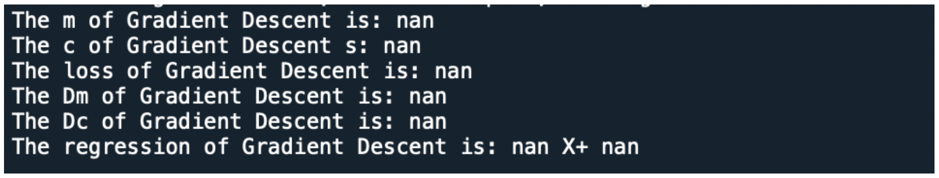
```
The m of Gradient Descent is: 5.40272148418424
The c of Gradient Descent s: 1.256613228402671
The loss of Gradient Descent is: 376.775879318383
The Dm of Gradient Descent is: 0.14490252399446324
The Dc of Gradient Descent is: -11.594429320840739
The regression of Gradient Descent is: 5.40272148418424 X+ 1.256613228402671
```



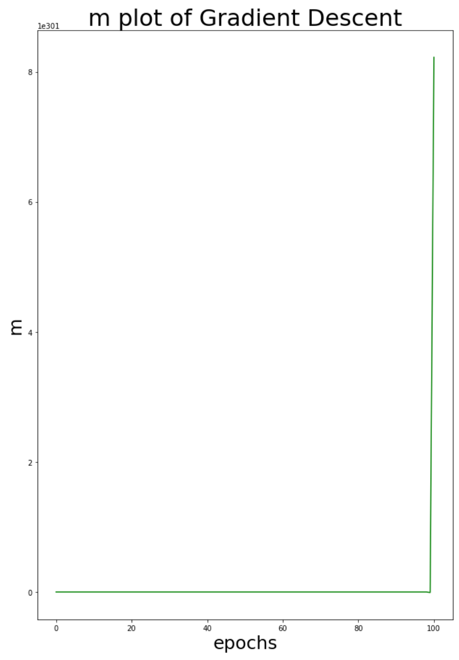
Here we can see that we are finding out very fast the optimal value of m, c and got the minimum value of the loss.
I thought It will take more iteration because the method over whole the data 1000 times and the steps (learning rate) were very small.
Another thing we can infer is that the graph is clean from noises as we described the model at the beginning.

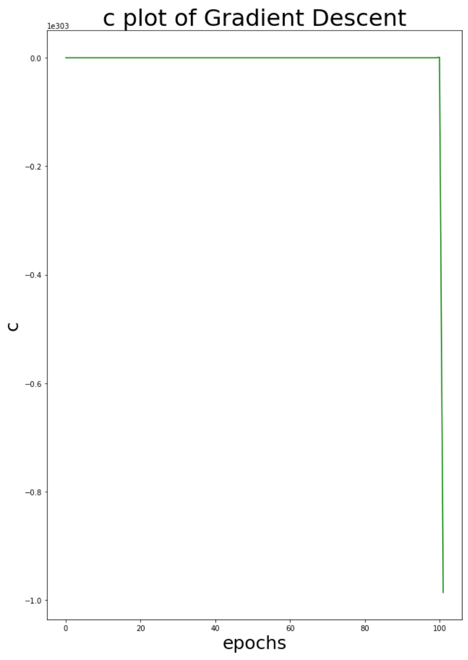
2. Learning rate= 0.1

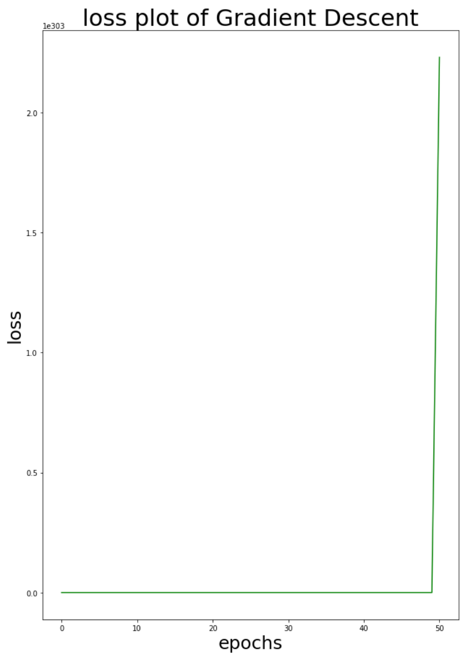
Results:



```
The m of Gradient Descent is: nan
The c of Gradient Descent s: nan
The loss of Gradient Descent is: nan
The Dm of Gradient Descent is: nan
The Dc of Gradient Descent is: nan
The regression of Gradient Descent is: nan X+ nan
```







The graph shows that the values of 'm' and loss get infinite and the 'c' get minus infinite after a few iterations.

We got nan values because of this (in Python after a very big/small value we got 'inf' and then 'nan').

I think the reason for these results is that the learning rate (step size) is too big, in this case every iteration causes the loss function to get a very big value and become infinite.

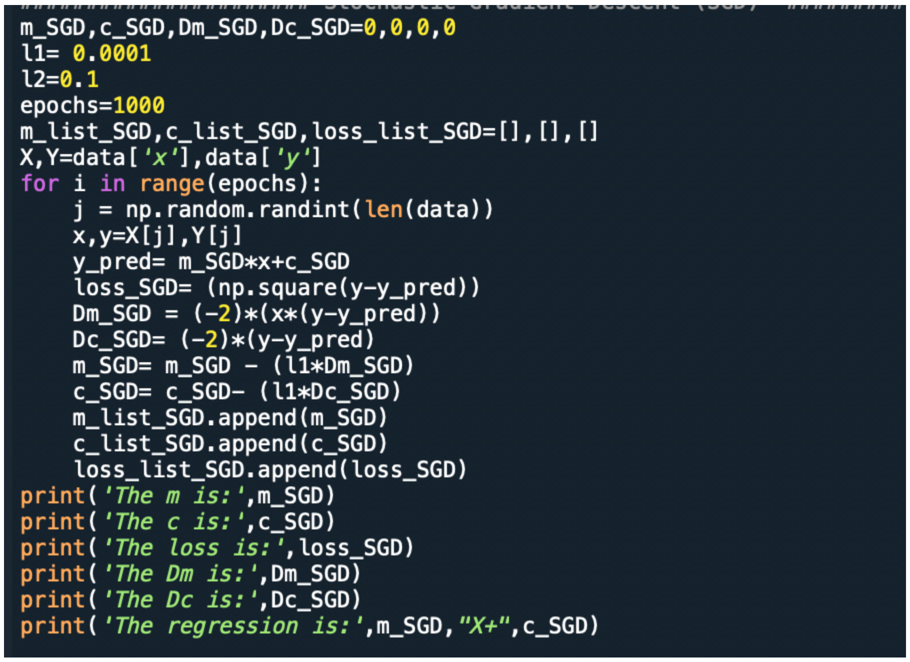
The 'm' and 'c' also get nan values because of the loss function.

The scenario is like this:

1. Learning rage (step size) is big.
2. Loss get infinite value.
3. Dm, Dc gets infinite value.
4. m, c gets an infinite value.
5. Our results are nan values.

I not expected to get these values because of the size of the step, I thought that we need a bigger step to get these infinities values.

Stochastic Gradient Descent (SGD) code:

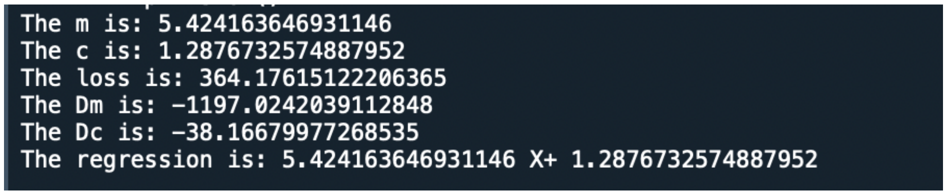


```
###################### Stochastic Gradient Descent (SGD) ##########
m_SGD,c_SGD,Dm_SGD,Dc_SGD=0,0,0,0
l1= 0.0001
l2=0.1
epochs=1000
m_list_SGD,c_list_SGD,loss_list_SGD=[],[],[]
X,Y=data['x'],data['y']
for i in range(epochs):
    j = np.random.randint(len(data))
    x,y=X[j],Y[j]
    y_pred= m_SGD*x+c_SGD
    loss_SGD= (np.square(y-y_pred))
    Dm_SGD = (-2)*(x*(y-y_pred))
    Dc_SGD= (-2)*(y-y_pred)
    m_SGD= m_SGD - (l1*Dm_SGD)
    c_SGD= c_SGD- (l1*Dc_SGD)
    m_list_SGD.append(m_SGD)
    c_list_SGD.append(c_SGD)
    loss_list_SGD.append(loss_SGD)
print('The m is:',m_SGD)
print('The c is:',c_SGD)
print('The loss is:',loss_SGD)
print('The Dm is:',Dm_SGD)
print('The Dc is:',Dc_SGD)
print('The regression is:',m_SGD,"X+",c_SGD)
```

In order to realize the stochastic part of the model, for each iteration I get a random number in the range of the data length, and the number will be the sample that the model will be trained on it, as we remember the model need every iteration to take only one sample.
The model repeats this over 1000 times like we defined the epochs.

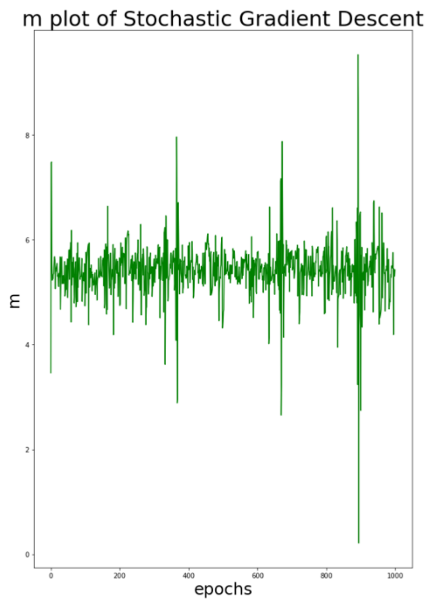
Note- if we want to change the learning rate, we need to replace l1 with l2.

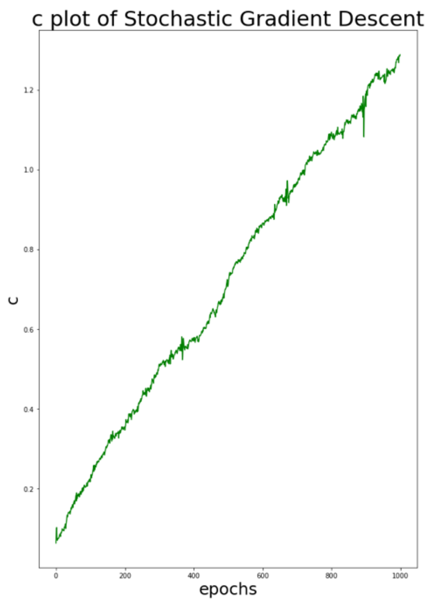
Stochastic Gradient Descent (SGD) graph:
1. Learning rate = 0.0001
Results:

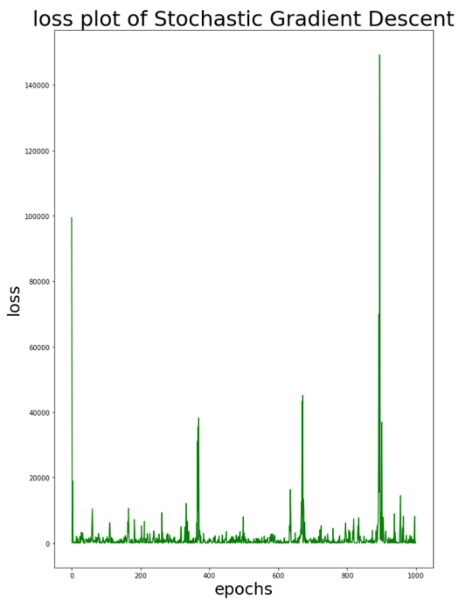


```
The m is: 5.424163646931146
The c is: 1.2876732574887952
The loss is: 364.17615122206365
The Dm is: -1197.0242039112848
The Dc is: -38.16679977268535
The regression is: 5.424163646931146 X+ 1.2876732574887952
```

As we can see from the results, the loss value is smaller than the Batch Gradient Descent, I was surprised by those results because I expected that the accuracy of the Batch Gradient Descent will be the best because he over all the data and in the SGD we take only 1000 random samples from the data.

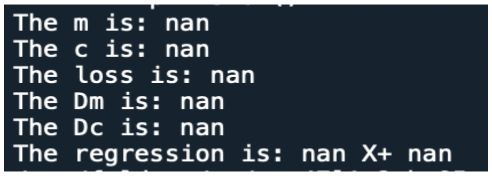






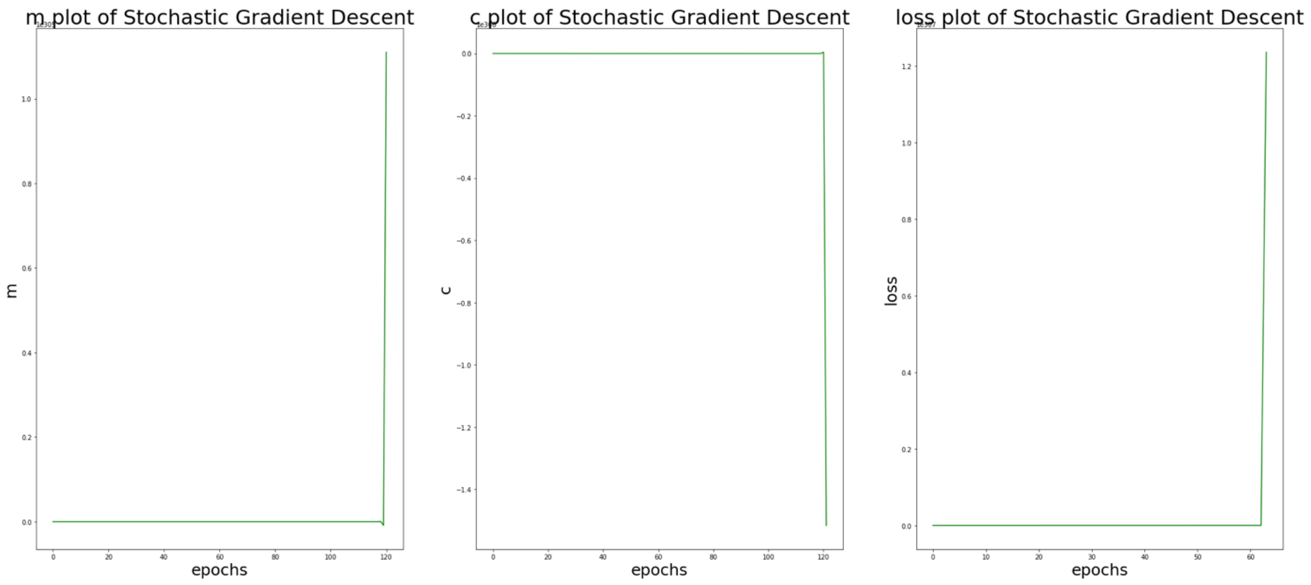
As we can see based on the 'm' and 'c' graphs they need more iteration to get the optimal values of them its make sense because every iteration of the model takes only 1 sample and this is a random sample.
Based on the loss graph we can see he is noisier compared to the Batch Gradient Descent I think this happened because in every iteration we get a random sample of the data and it can cause biased results.

2. **Learning rate = 0.1**
Results:

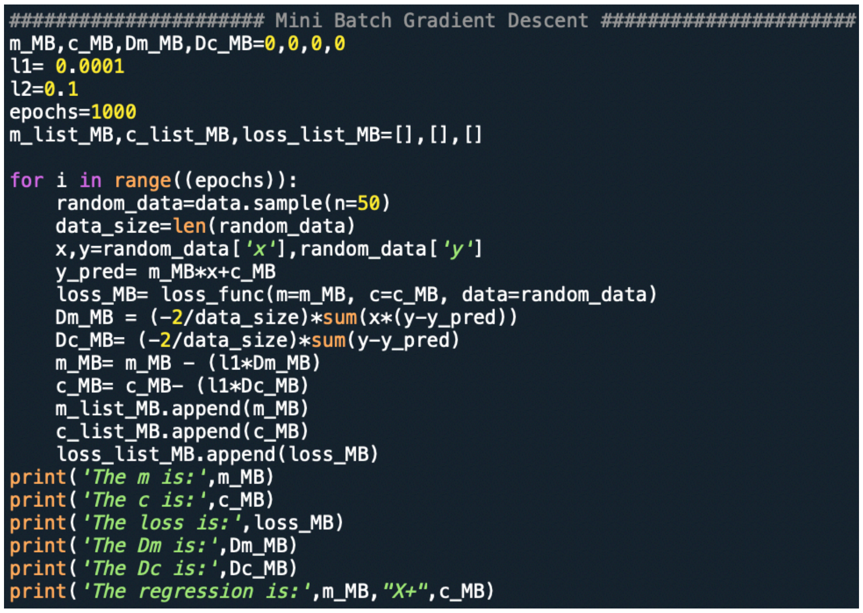


```
The m is: nan
The c is: nan
The loss is: nan
The Dm is: nan
The Dc is: nan
The regression is: nan X+ nan
```

The same results as the Batch Gradient Descent here.



Mini Batch Gradient Descent code:



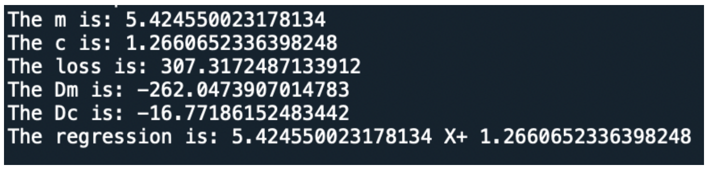
```
####################### Mini Batch Gradient Descent ######################
m_MB,c_MB,Dm_MB,Dc_MB=0,0,0,0
l1= 0.0001
l2=0.1
epochs=1000
m_list_MB,c_list_MB,loss_list_MB=[],[],[]

for i in range((epochs)):
    random_data=data.sample(n=50)
    data_size=len(random_data)
    x,y=random_data['x'],random_data['y']
    y_pred= m_MB*x+c_MB
    loss_MB= loss_func(m=m_MB, c=c_MB, data=random_data)
    Dm_MB = (-2/data_size)*sum(x*(y-y_pred))
    Dc_MB= (-2/data_size)*sum(y-y_pred)
    m_MB= m_MB - (l1*Dm_MB)
    c_MB= c_MB- (l1*Dc_MB)
    m_list_MB.append(m_MB)
    c_list_MB.append(c_MB)
    loss_list_MB.append(loss_MB)
print('The m is:',m_MB)
print('The c is:',c_MB)
print('The loss is:',loss_MB)
print('The Dm is:',Dm_MB)
print('The Dc is:',Dc_MB)
print('The regression is:',m_MB,"X+",c_MB)
```

In this model in order to save the randomness I chose to use "df sample()" function, I determine the batch size from the data and the function returns randomness data in this size.
The method over the **new** data and make the calculation of GD in order to find the linear regression equation.
Every iteration the model gets a new batch but in the same size.
Randomly I defined batch size as 50 samples.
In order to check myself I try to define the batch size as the length of the data in that case the model is exactly like Batch Gradient Descent (because the model is over 1000 times the whole data) and indeed I got the same results.

Mini Batch Gradient Descent graph:

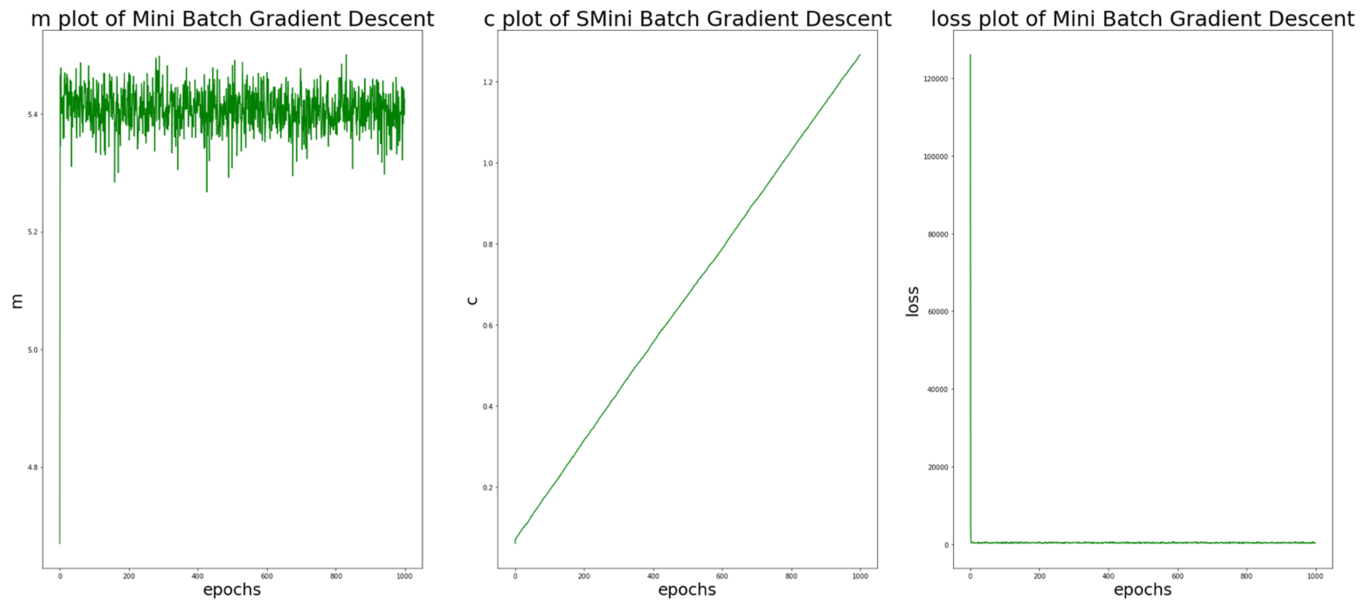
1. Learning rate = 0.0001

Results:



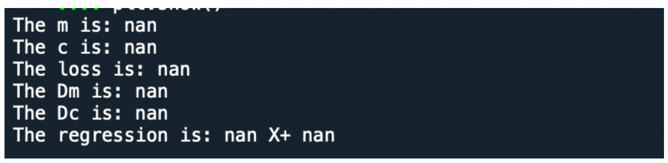
```
The m is: 5.424550023178134
The c is: 1.2660652336398248
The loss is: 307.3172487133912
The Dm is: -262.0473907014783
The Dc is: -16.77186152483442
The regression is: 5.424550023178134 X+ 1.2660652336398248
```

We see that the loss value is the smallest value compare to other models.



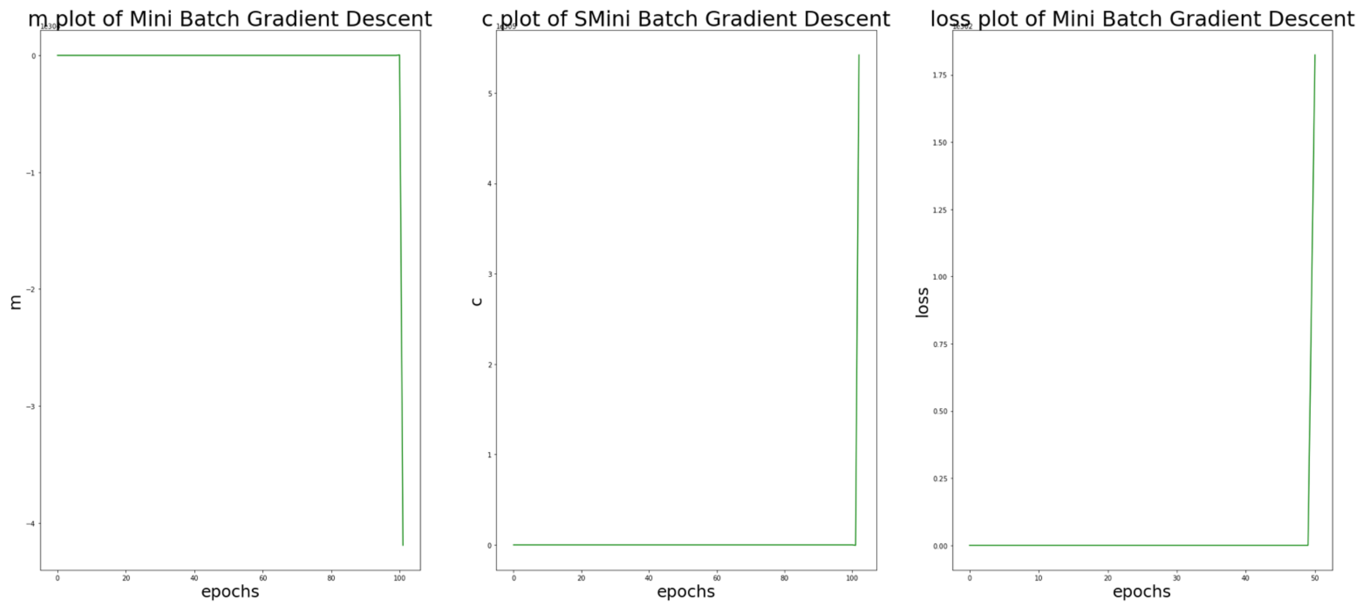
As we can see based on the 'm' and 'c' graphs they need less iteration to get the optimal values of them (compared to SGD).
We can see that the 'm' and 'c' graphs are less noisy.
Based on the loss graph we can see he is also less noisy compared to the SGD.
The model receives its optimum values faster.
This model his combines concepts from both batch gradient descent and stochastic gradient descent and we see it by the accuracy of Batch Gradient Descent and little noise of SGD.

1. **Learning rate = 0. 1**
Results:



```
The m is: nan
The c is: nan
The loss is: nan
The Dm is: nan
The Dc is: nan
The regression is: nan X+ nan
```

The same results as the Batch Gradient Descent and SGD here.



Conclusion:

We tested all types of Gradient Descent with two different learning rates (0.1,0.0001).
If we haven't a problem with accuracy and we care only about time I think that the SGD is the model we will choose.
But, if we have an issue of time and we want good accuracy I think that in this case, the best model is the Mini Batch Gradient Descent.
For any model we will work with, we need to pay attention to the learning rates, because inappropriate learning rates will give us inappropriate results.
If we will choose a too large learning rate we will miss the optimal point and if we will choose a too small learning rate its will take us a long time to get the results.