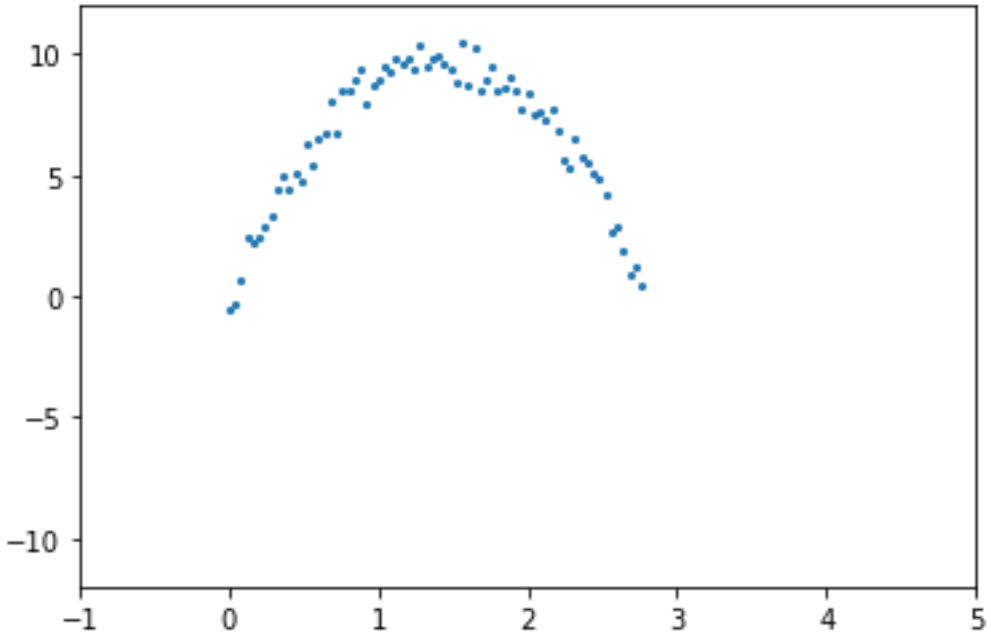
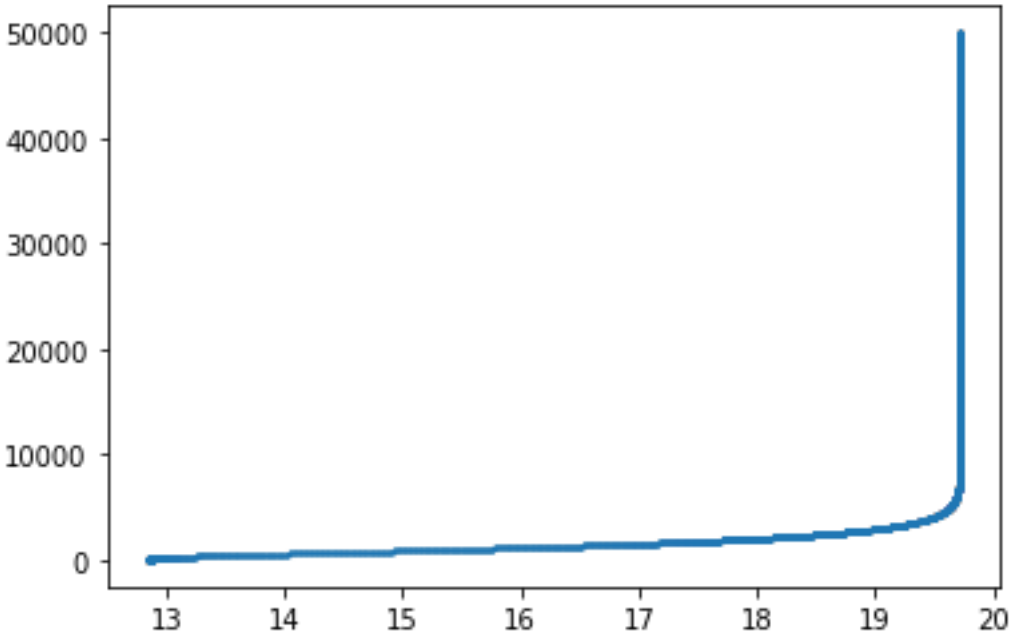
ASSIGNMENT – 2

1.1) Here, I have plotted the training data i.e., time(x axis) vs height(y axis).



1.2) This is the plot for cost history (J) vs number of iterations for Batch Gradient Descent. (where alpha = 0.01 and number of iterations = 50000)

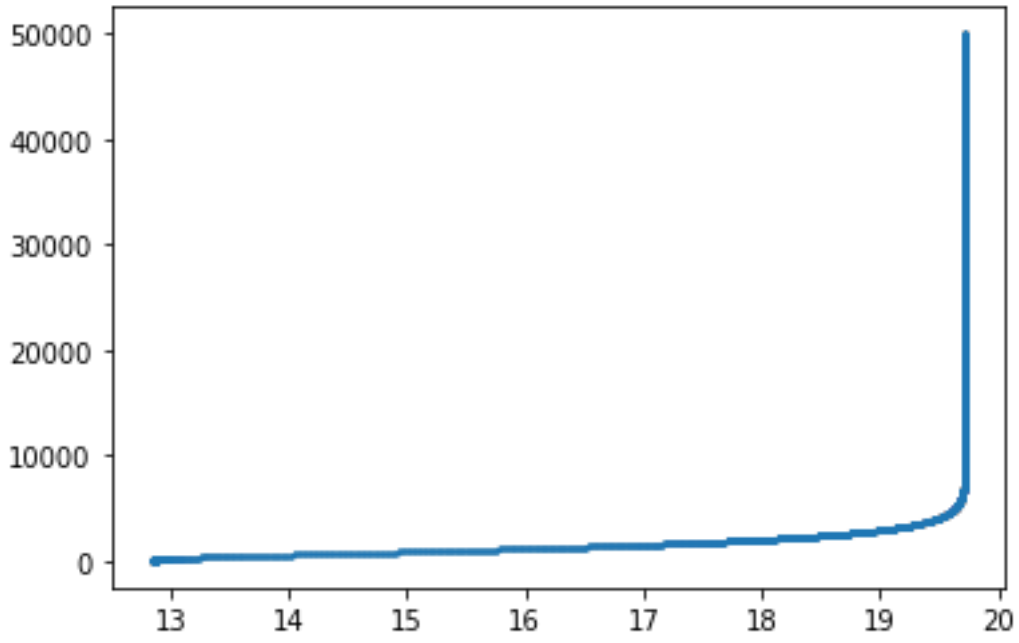


Code snippet of part 2

```
    # This function optimizes the weights w_0, w_1, w_2. Batch Gradient Descent method
    def BgradientDescent(self, x, y, w, alpha, iters):
      m = len(x)  # number of training examples
      w_orig = w.copy() # To keep a copy of original weights

      J_history = []   # Use a python list to save cost in every iteration

      for i in range(iters):
        # Loop to update weights (w vector)
        # Also save cost at every step
        w = np.array(w)
        w = w - alpha * np.array(self.grad(x,y,w))  # NOT SURE if we should use w or w_orig while calling grad fxn.
        J_history.append(self.computeCost(x,y,w))

      return w, J_history
```

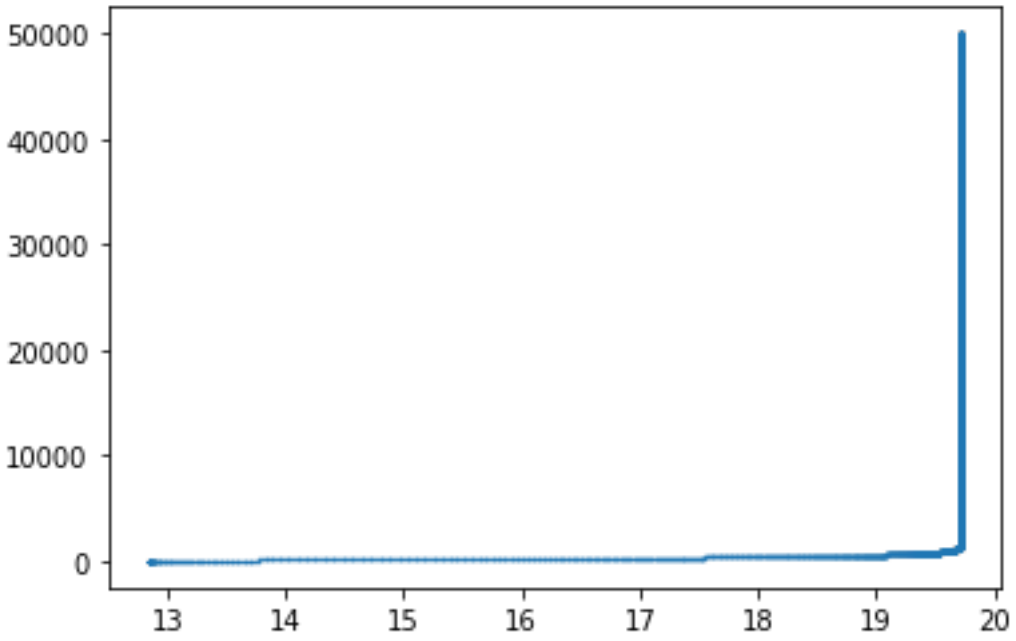
```
model = nlr()
# model.test()
alpha = [0.1, 0.5, 0.01, 0.05]
w = [1,1]
a,b = model.BgradientDescent(x, y, w, alpha[3] , 50000)
plt.plot(b,[i for i in range(50000)],'o',markersize=0.5)
```

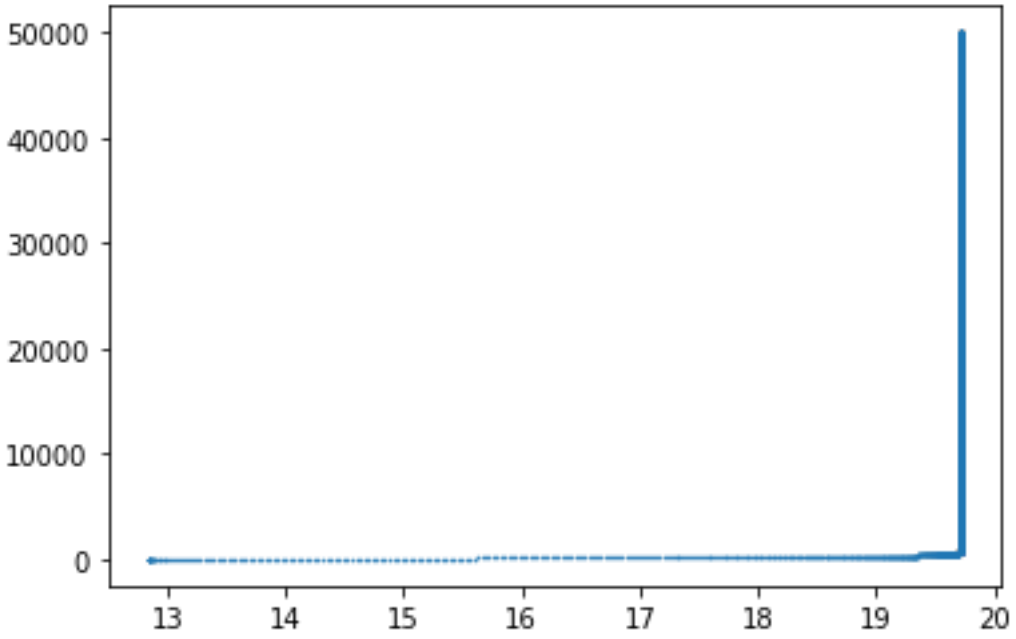
This is the plot for cost history (J) vs number of iterations for Stochastic Gradient Descent. (where alpha = 0.01 and number of iterations = 50000)

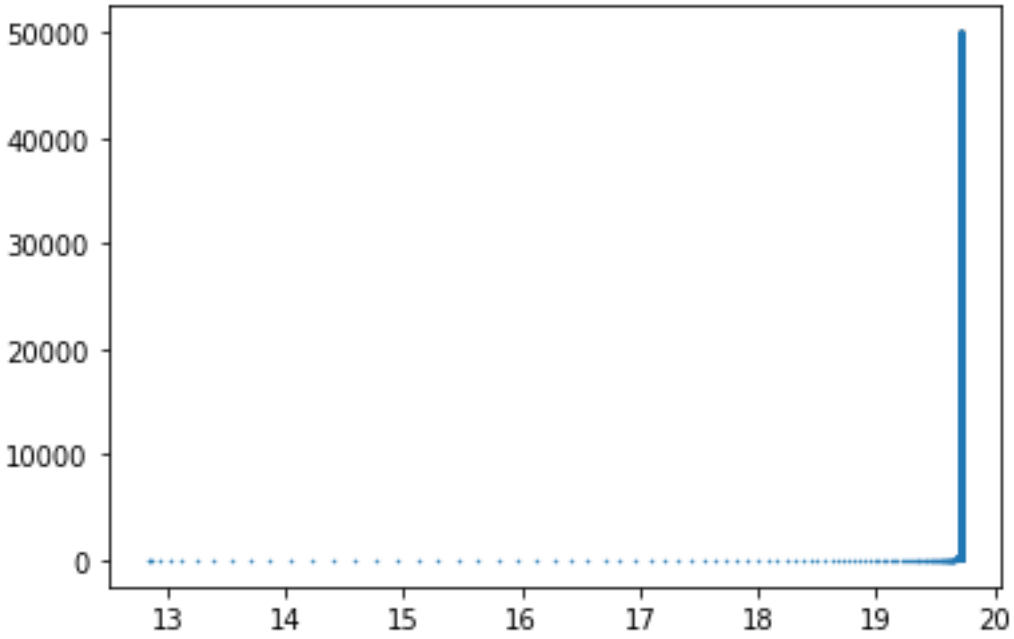


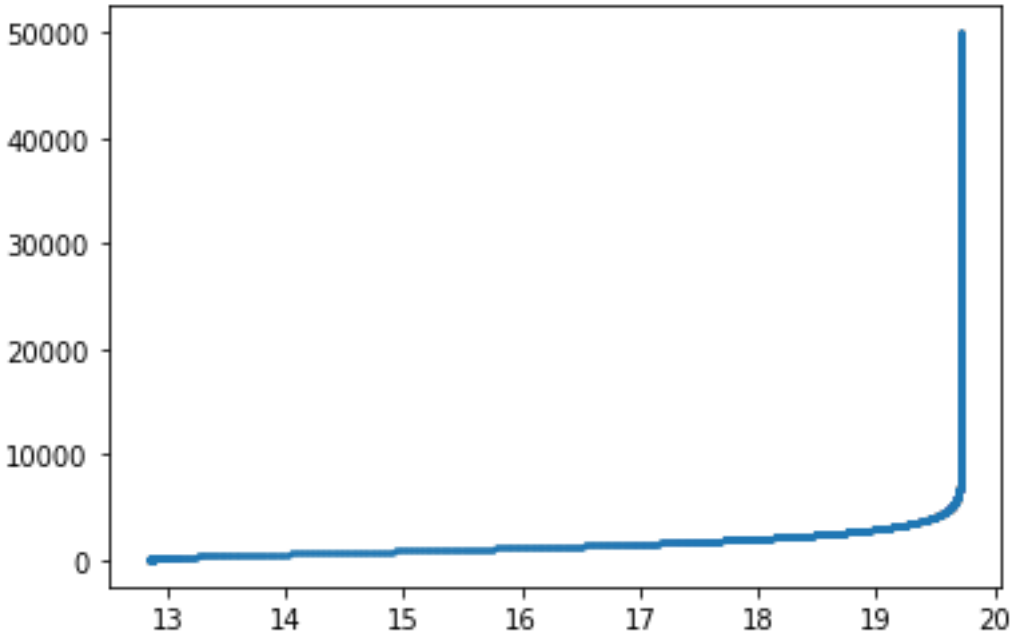
We can see that there is not much difference in the 2 plots , they are very similar except a very slight difference in slopes.

1.3) This is the plot for cost history (J) vs number of iterations for different learning rates ($\alpha = 0.05, 0.1, 0.5, 0.01$ respectively).
From these plots we can infer that for a greater value of alpha, the weights w are optimized faster and in lesser number of iterations.






50000
40000
30000
20000
10000
0
13
14
15
16
17
18
19
20


50000
40000
30000
20000
10000
0
13
14
15
16
17
18
19
20

2.1) Code to perform data cleaning and mean normalization of features.

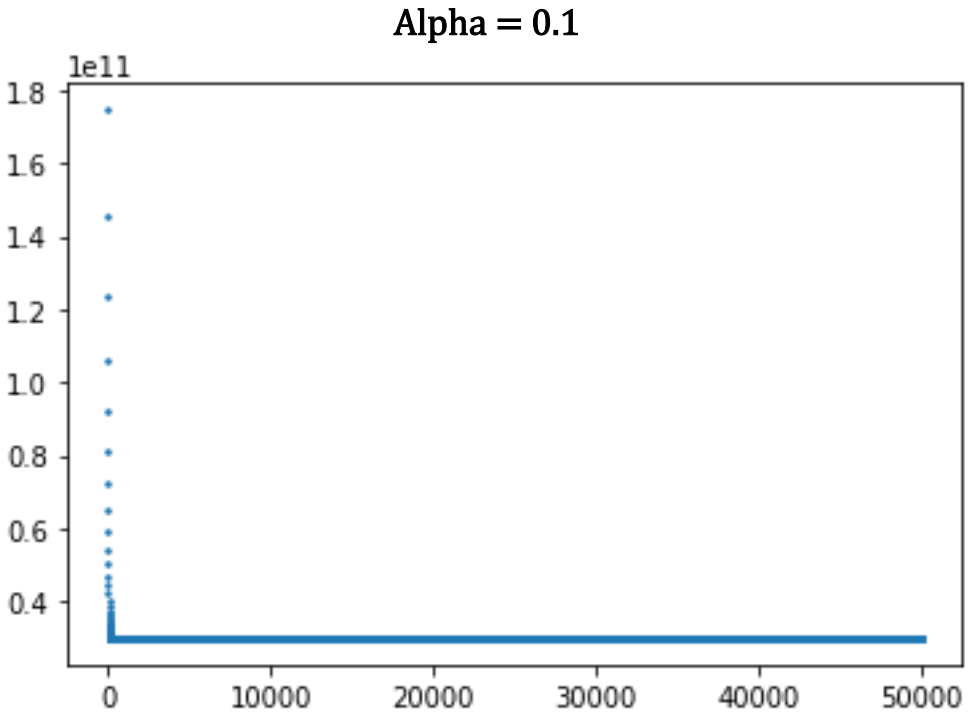
```
data =  pd.read_csv('prob2data.csv')
y = np.array(data['price'].values)
# print(y.shape)

x0 = np.ones(y.shape[0])
x1 = np.array(data['bedrooms'].values)
x2 = np.array(data['bathrooms'].values)
x3 = np.array(data['sqft_living'].values)
x4 = np.array(data['floors'].values)
x5 = np.array(data['yr_built'].values)

X = np.stack([x1,x2,x3,x4,x5],axis=1)
mean = np.array([np.sum(X[:,i])/X.shape[0] for i in range(5)])
sd = np.std(X,axis=0)
x = np.append((np.array([x0])).T, np.array(([X[i]-mean for i in range(y.shape[0])])/sd),axis=1)
```

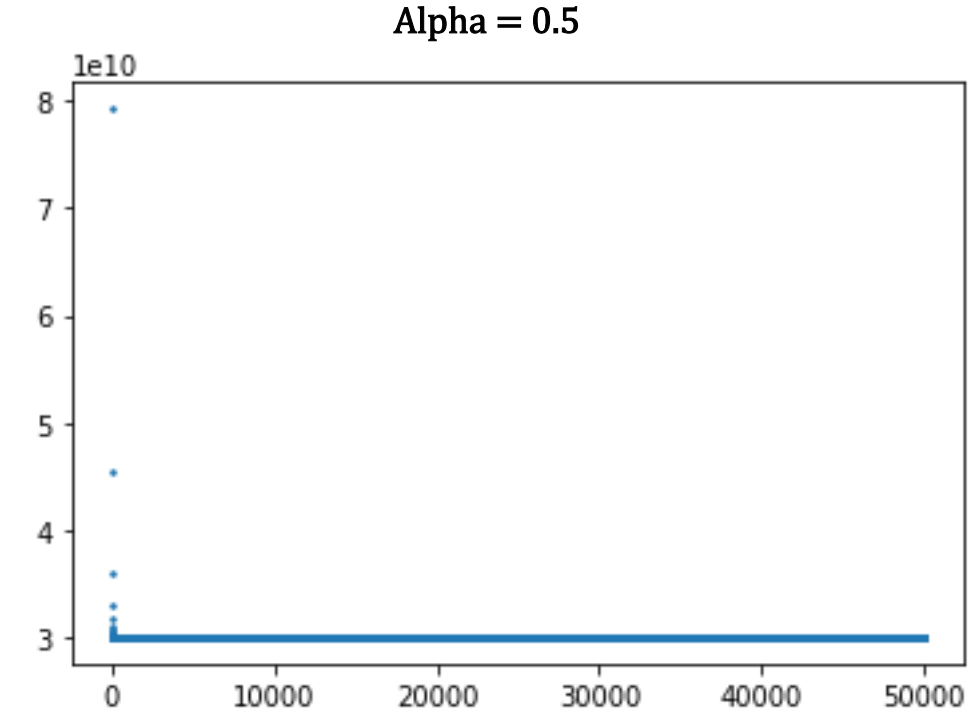
2.3) Code snippet for predicting the price for the sample case given.

```
l = [4,2.5,2570,2,2005]
l = np.append((np.array([1])),(np.array(l-mu)/sigma))
a,b = self.bgdMulti(x,y,w,0.01,50000)
price =  np.dot(l,a.T)
```

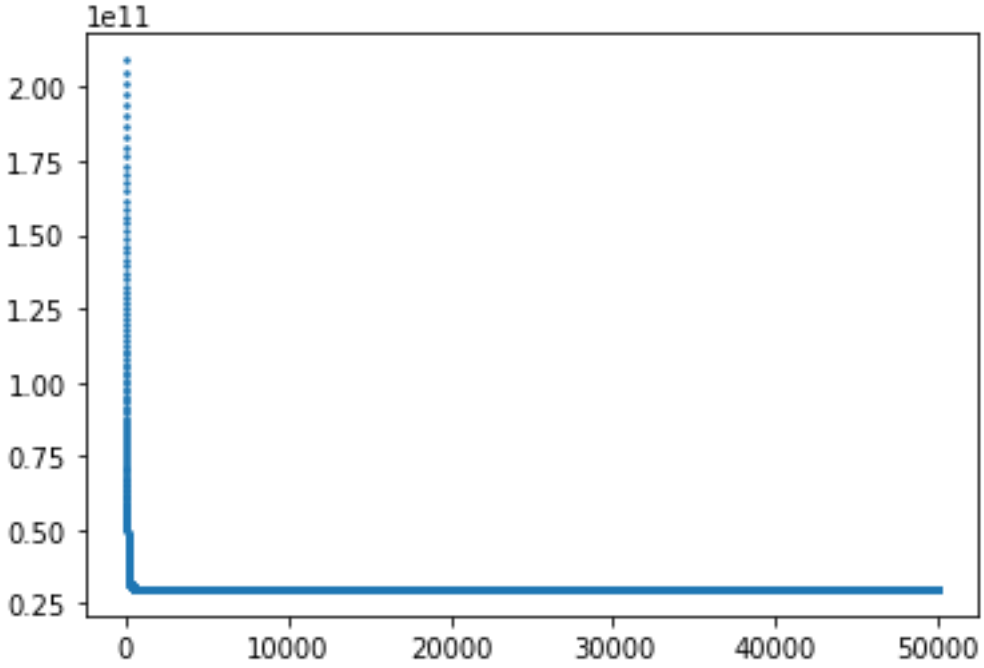
The price obtained from the model is : `584161.4194725453`
Hence, the percentage error between the predicted cost and actual cost will be calculated as :-
% error = ((719000 – 584161.4194725453)/ 584161.4194725453)*100
= **23.0824%**

2.4)

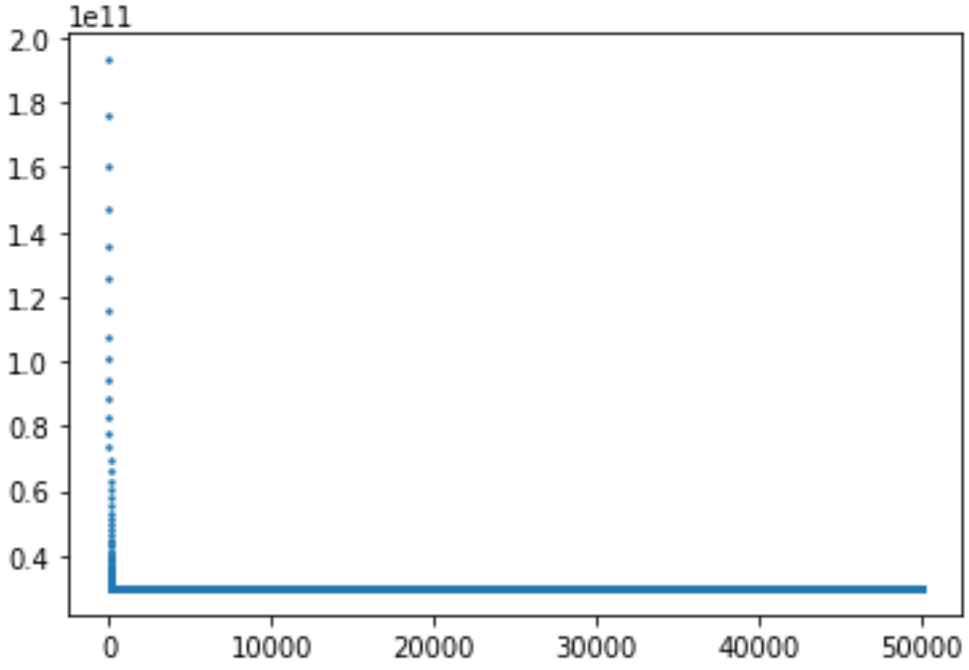

Alpha = 0.1
1e11
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0
10000
20000
30000
40000
50000


Alpha = 0.5
1e10
8
7
6
5
4
3
0
10000
20000
30000
40000
50000

Alpha = 0.01



Alpha = 0.05



From these plots we can infer that for a greater value of alpha, the weights w are optimized faster and in lesser number of iterations. Here, for the value of alpha $= 0.5$, the loss function is minimized the fastest as compared to when alpha $= 0.01$