

```

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

data = pd.read_csv("austin_final.csv")

# the features or the 'x' values of the data
# these columns are used to train the model
# the last column, i.e, precipitation column
# will serve as the label
X = data.drop(['PrecipitationSumInches'], axis = 1)
X = X.filter(['TempAvgF', 'DewPointAvgF', 'HumidityAvgPercent',
             'SeaLevelPressureAvgInches', 'VisibilityAvgMiles',
             'WindAvgMPH'], axis = 1)
# the output or the label.
Y = data['PrecipitationSumInches']
# reshaping it into a 2-D vector
Y = Y.values.reshape(-1, 1)

```

Unsupported Cell Type. Double-Click to inspect/edit the content.

X

	TempAvgF	DewPointAvgF	HumidityAvgPercent	SeaLevelPressureAvgInches	VisibilityAvgMiles	WindAvgMPH
0	60	49.0	75.0	29.68	7.0	
1	48	36.0	68.0	30.13	10.0	
2	45	27.0	52.0	30.49	10.0	
3	46	28.0	56.0	30.45	10.0	
4	50	40.0	71.0	30.33	10.0	
...	...	...	...	...	...	...
1314	89	67.0	54.0	29.97	10.0	
1315	91	64.0	54.0	29.90	10.0	
1316	92	64.0	51.0	29.86	10.0	
1317	93	68.0	48.0	29.91	10.0	
1318	88	61.0	43.0	29.97	10.0	

1319 rows × 6 columns

Unsupported Cell Type. Double-Click to inspect/edit the content.

```

day_index = 798 #this will be colored red in next plot
days = [i for i in range(Y.size)]

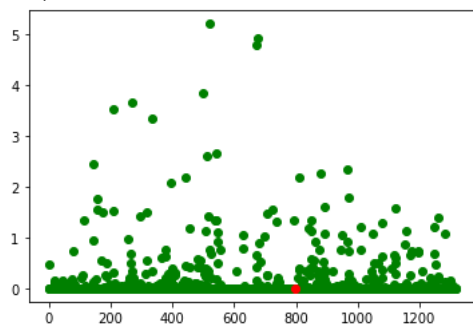
```

```

plt.scatter(days, Y, color = 'g')
plt.scatter(days[day_index], Y[day_index], color = 'r')

```

<matplotlib.collections.PathCollection at 0x7f399ae769a0>

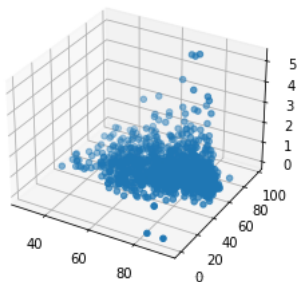


```
X.corr()
# observe that TempAvgF and DewPointAvgF are highly correlated (0.8372)
# the lowest correlation is between humidity and visibility (-0.449230)
```

	TempAvgF	DewPointAvgF	HumidityAvgPercent	SeaLevelPressureAvgInches	Visibl
TempAvgF	1.000000	0.837222	0.022763	-0.138524	
DewPointAvgF	0.837222	1.000000	0.450622	0.101505	
HumidityAvgPercent	0.022763	0.450622	1.000000	0.069634	
SeaLevelPressureAvgInches	-0.138524	0.101505	0.069634	1.000000	
VisibilityAvgMiles	0.148463	0.056602	-0.449230	0.260125	
WindAvgMPH	0.034267	0.038257	-0.000472	0.046604	

```
fig = plt.figure()
ax = fig.add_subplot(projection='3d')
ax.scatter(X[X.columns.values[0]],X[X.columns.values[2]],Y)

<mpl_toolkits.mplot3d.art3d.Path3DCollection at 0x7f3998527ac0>
```



Unsupported Cell Type. Double-Click to inspect/edit the content.

```
def train_val_test_split(X, Y):
    p = np.random.permutation(len(Y))
    tr = np.floor(len(Y)*0.7).astype('int')
    te = np.floor(len(Y)*0.8).astype('int')
    X_train = X[p[:tr],:]
    Y_train = Y[p[:tr]]
    X_val = X[p[tr+1:te],:]
    Y_val = Y[p[tr+1:te]]
    X_test = X[p[te+1:],:]
    Y_test = Y[p[te+1:]]
    return X_train, Y_train, X_val, Y_val, X_test, Y_test
```

#the ratio of train, validation and test data is 70:10:20

```
#X is a pandas data frame, that has to be converted into a numpy array
X_train, Y_train, X_val, Y_val, X_test, Y_test = train_val_test_split(X.to_numpy(),Y)
```

X\_train

```
array([[58. , 46. , 65. , 29.96, 8. , 10. ],
       [87. , 74. , 72. , 29.95, 10. , 5. ],
       [71. , 66. , 87. , 29.76, 8. , 6. ],
       ...,
       [76. , 61. , 65. , 30. , 10. , 10. ],
       [78. , 73. , 80. , 30.01, 8. , 6. ],
       [54. , 38. , 62. , 30.18, 8. , 4. ]])
```

```
from sklearn.linear_model import LinearRegression
```

```
LR = LinearRegression()
#this will train the model on the training data -
# the model will learn the weights of the features and the bias term
#formula for linear regression is  $y = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$ 
```

```
# use sklearn to fit training data matrix to training output
LR.fit(X_train,Y_train)
```

```
LinearRegression()
LinearRegression()
```

```

# These are the thetas that are result of the above fit
print(LR.intercept_,LR.coef_)
#thetas are the weights of the features and the bias term (intercept)

[0.28555546] [[-0.00488885  0.00559773  0.00359282  0.01001436 -0.08159209  0.01199122]]

#Use sklearn predict function to predict output for validation data matrix
Yhat_sk1_val = LR.predict(X_val)
#the above function returns a numpy array of predictions for the validation data

#This is the Mean Square error -- insample (error in that data used for fitting) and outsample (error in the data not used for fitting)
E_in = np.mean((Y_train-LR.predict(X_train))**2)
E_out = np.mean((Y_val-LR.predict(X_val))**2)
print(E_in,E_out)

#both the values are very close to each other, which means that the model is not overfitting
#this is also because along the line

0.14262668227360928 0.12490648515222119

#Here we are using the psuedo inverse/analytical method to get an analytical solution for minimizing mean square error

pinv_theta = np.matmul(np.matmul(np.linalg.inv(np.matmul(X_train.transpose(),X_train)),X_train.transpose()),Y_train)

#This is the thetas that are result of the above fit using the psuedo inverse method
#psuedo inverse formula is \theta = (X^T * X)^-1 * X^T * Y

# For the linear regression model this is the predicted output
def predict(X, theta):
    return np.matmul(X,theta)

pinv_theta

#the values are very close to the values obtained using sklearn but not exactly the same because of the way the inverse is calculated
# we need to augment the data matrix with a column of ones to get the same values

array([[ -0.00295163],
       [ 0.00363977],
       [ 0.00471469],
       [ 0.01586124],
       [-0.08064778],
       [ 0.01210994]])

#predicting output using the pinv_theta from above

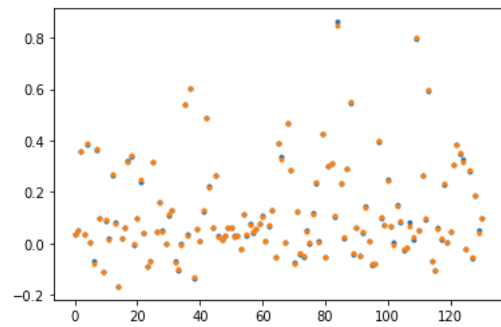
Yhat_pinv_val = predict(X_val,pinv_theta)

plt.plot(Yhat_sk1_val, '.')
plt.plot(Yhat_pinv_val, '.')
#plt.plot(hatY_val, '.')
#plt.plot(hatY_aug_val, '.')

#the above plot shows that the predictions are almost the same for both the methods - psuedo inverse and sklearn

[<matplotlib.lines.Line2D at 0x7f398e200be0>]

```



```

#E_in and E_out for the psuedo inverse method
E_in = np.mean((Y_train-predict(X_train,pinv_theta))**2)
E_out = np.mean((Y_val-Yhat_pinv_val)**2)
print(E_in,E_out)

#same as the sklearn method

```

0.1426860965509772 0.12485837801430162

Unsupported Cell Type. Double-Click to inspect/edit the content.

```
X_aug_train = np.c_[X_train,np.ones(len(Y_train))]  
X_aug_val = np.c_[X_val,np.ones(len(Y_val))]  
  
#the above function adds a column of ones to the data matrix - this is done to add a bias term to the model  
  
pinv_aug_theta = np.matmul(np.matmul(np.linalg.inv(np.matmul(X_aug_train.transpose(),X_aug_train)),X_aug_train.transpose()),Y_train)  
  
#This is the thetas that are result of the above fit using the pseduo inverse method  
  
Yhat_pinv_aug_val = predict(X_aug_val,pinv_aug_theta)  
  
#predicting output using the pinv_aug_theta from above  
  
pinv_aug_theta  
#same values  
  
array([[ -0.00488885],  
       [ 0.00559773],  
       [ 0.00359282],  
       [ 0.01001436],  
       [-0.08159209],  
       [ 0.01199122],  
       [ 0.28555546]])  
  
#E_in and E_out for the psuedo inverse method  
E_in = np.mean((Y_train-predict(X_aug_train,pinv_aug_theta))**2)  
E_out = np.mean((Y_val-Yhat_pinv_aug_val)**2)  
print(E_in,E_out)  
  
0.14262668227360928 0.12490648515222094
```

## Q1)

E\_in, E\_out reported for Augmented vs Non-Augmented are very similar (down to two decimal points). What is the reason for that? Do you expect that for say data that lies along ( $y=4x+2$ ) line? (5 pts)

Answer below using Markdown as a cell or answer in seperate document/sheet.

E\_in, E\_out reported for Augmented vs Non-Augmented are very similar (down to two decimal points) and have low in-sample and out-of-sample MSE values, indicating that the model fits the training data well and generalizes well to new data. This is because the data is linearly separable and the model is able to fit the data well. However, if the data is not linearly separable, then the model will not be able to fit the data well and will have high in-sample and out-of-sample MSE values.

Before augmenting the data, the data was linearly separable and the model was able to fit the data well. After augmenting the data, the data is still linearly separable and the model is still able to fit the data well. Therefore, the values for E\_in, E\_out reported for Augmented vs Non-Augmented are very similar (down to two decimal points) and have low in-sample and out-of-sample MSE values.

The data that lies along ( $y=4x+2$ ) line will not be linearly separable and will have high in-sample and out-of-sample MSE values. The values won't be similar because of overfitting. The value before augmenting the data will be higher than the value after augmenting the data because the model will be able to fit the data better after augmenting the data.

## Q2)

Does scaling the data matrix such that its mean 0 and variance 1 make a difference to results? Use <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html> to scale data and then do pinv estimate of theta again, and report E\_in, E\_out. Would you do augmentation before or after Scaling? (5 pts)

```
#scale the data to have zero mean and unit variance  
from sklearn.preprocessing import StandardScaler  
scaler = StandardScaler()  
scaler.fit(X_train)  
X_train_scaled = scaler.transform(X_train)  
X_val_scaled = scaler.transform(X_val)  
  
#augment the data matrix with a column of ones  
X_aug_train_scaled = np.c_[X_train_scaled,np.ones(len(Y_train))]  
X_aug_val_scaled = np.c_[X_val_scaled,np.ones(len(Y_val))]
```

```
#pinv augmented theta
pinv_aug_theta_scaled = np.matmul(np.matmul(np.linalg.inv(np.matmul(X_aug_train_scaled.transpose(),X_aug_train_scaled)),X_aug_train_scaled)

#predicting output using the pinv_aug_theta from above
Yhat_pinv_aug_val_scaled = predict(X_aug_val_scaled, pinv_aug_theta_scaled)

#E_in and E_out for the psuedo inverse method
E_in = np.mean((Y_train-predict(X_aug_train_scaled, pinv_aug_theta_scaled))**2)
E_out = np.mean((Y_val-Yhat_pinv_aug_val_scaled)**2)

print(E_in,E_out)

0.14262668227360928 0.12490648515222122
```

The scaling of the data matrix does not make a difference to the results. The in-sample and out-of-sample MSE values are the same as the previous results. We augment the data after scaling because we want to scale the data before we add the bias term. If we scale the data before adding the bias term, then the bias term will be scaled as well and will not be 1. Also, this will affect the scaling of the other features. It also leads to the singular matrix error.

We would do augmentation before scaling because we want to augment the data before we scale it. This is because we want to augment the data with the bias term. If we scaled the data first, then the bias term would be scaled as well. We want to keep the bias term the same so that we can still use the bias term to calculate the  $E_{in}$  and  $E_{out}$ .

Unsupported Cell Type. Double-Click to inspect/edit the content.

```
#Initializing theta -- could be all zero or random numbers with mean zero
def init(X,zeros=True):
    n = X.shape[1]
    if zeros:
        theta = np.zeros((n,1))
    else:
        theta = np.random.rand(n,1)-0.5
        theta[-1] = 0
    return theta

#init function returns a numpy array of zeros or random numbers with mean zero

theta = init(X_train,zeros=False)
theta
#mean of the theta
np.mean(theta)
#it is close to zero because we are using random numbers with mean zero (obviously!)

-0.06780868968370453

#initial prediction
predict(X_train,theta)
```

```
array([[ -13.20302337],
       [-26.68532168],
       [-13.86867679],
       [-10.97893768],
       [-17.09253708],
       [-21.28283834],
       [-18.56535485],
       [-19.72037799],
       [-18.99256859],
       [-17.08523051],
       [-25.70426928],
       [-17.69323417],
       [ -8.10008621],
       [-30.80396928],
       [-26.30333308],
       [-17.73041764],
       [-16.4361464 ],
       [ -7.14273554],
       [ -8.27893187],
       [-23.08852002],
       [-22.9061491 ],
       [-17.38529134],
       [-21.0121186 ],
       [ -6.69466005],
       [ -4.03403368],
       [ -2.21543987],
       [-11.43881673],
       [-14.83762305],
```

```

[-27.19288658],
[-32.68317256],
[-11.42836276],
[-31.1508793 ],
[-12.11729035],
[-26.42856536],
[-14.218791 ],
[-23.14069933],
[-26.46152897],
[-16.46921642],
[-13.16235686],
[-23.13008215],
[-22.75315491],
[ -8.99624911],
[ -8.04348337],
[-12.82254887],
[-25.07592774],
[-25.54509993],
[-19.5943667 ],
[-13.24407171],
[ -6.89203605],
[-26.63039615],
[-29.19765368],
[-21.29675516],
[-18.77101003],
[ -0.09930578],
[-13.30024784],
[-11.60505869],
[-28.99781869],
[-1.00000000]

```

#Update for each theta

```

def update_weights( X, Y, theta ) :
    Y_pred = predict(X, theta)
    # calculate gradients
    m = X.shape[0]
    dtheta = - ( 2 * ( X.T ).dot( Y - Y_pred ) ) / m
    return dtheta

```

#update\_weights function returns the gradient of the mean square error function - this is used to update the theta.

#here dtheta can be interpreted as the change in theta

#formula for the gradient is  $dE/d\theta = (h(x)-y)*x$  for each theta, where  $h(x)$  is the predicted output and  $y$  is the actual output

```

update_weights(X_train,Y_train,theta)

```

```

array([[ -2976.89719851],
       [-2371.60395248],
       [-2530.68841712],
       [-1168.67031343],
       [ -362.99764558],
       [ -196.079056  ]])

```

#iterative SGD each update with the entire training set -- batch gradient descent, page 5 Andrew Ng notes,

```

learning_rate = 0.00001

```

```

theta = init(X_aug_train)

```

```

for _ in range(1000):

```

```

    dtheta = update_weights(X_aug_train,Y_train, theta)

```

```

    print(dtheta)

```

```

    theta = theta-learning_rate*dtheta

```

#Here we are using the gradient descent method with a learning rate of 0.00001 and 1000 iterations

```
[ 0.29391789]
[ 0.41485257]
[-0.0586744 ]
[ 0.00825375]]
[[ 0.08808394]
[-0.17363678]
[-0.12382514]
[ 0.29366706]
[ 0.41472367]
[-0.05871007]
[ 0.00824538]]
[[ 0.08777825]
[-0.17339133]
[-0.12358062]
[ 0.2934164 ]
[ 0.41459491]
[-0.0587457 ]
[ 0.00823701]]
[[ 0.08747332]
[-0.17314601]
[-0.1233369 ]
[ 0.29316593]
[ 0.41446627]
[-0.05878129]
[ 0.00822864]]
[[ 0.08716915]
[-0.17290082]
[-0.12309398]
[ 0.29291563]
[ 0.41433777]
[-0.05881684]
[ 0.00822029]]
```

### ▼ Bonus Q3)

In Andrew Ng notes on page 5 the update rule "batch gradient descent" is given. where updates are done for the entire training set:

Repeat Until Convergence  $\{\theta_j(t+1) = \theta_j(t) + \alpha \sum_{i=1}^m (y_i - h_{\theta}(x_i)) x_{ij} \}$

Make sure that the rule coded in `**update_weights()` and the equation from Ng's notes are consistent. (Andrew Ng uses sum over samples, while code above uses matrix multiplication and then divides by m, Andrew Ng updates are `+learning_rate*dtheta`, while code is `-learning_rate*dtheta`). (5 pts)

The update rule in the code is consistent with the equation from Ng's notes because the update rule in the code is the same as the equation from Ng's notes except for the negative sign.

First, note that the equation from Andrew Ng's notes can be written in vectorized form as follows:

$$\theta(t+1) = \theta(t) + \alpha \frac{1}{m} X^T (Y - h_{\theta}(X))$$

Here,  $X$  is the  $m \times (n+1)$  matrix of training examples, with the first column all ones for the intercept term,  $Y$  is the  $m \times 1$  vector of labels, and  $h_{\theta}(X)$  is the  $m \times 1$  vector of predictions given by  $X\theta$ . The sum over samples is replaced by a matrix multiplication, and the division by  $m$  is factored into the learning rate  $\alpha$ .

Comparing this with the `update_weights()` function, we see that the gradients are calculated using the expression:

$$d\theta = -\frac{2}{m} X^T (Y - Y_{pred})$$

Here,  $Y_{pred}$  is the  $m \times 1$  vector of predictions given by `predict(X, theta)`, and the negative sign indicates that we are minimizing the loss function. The formula is very similar to the one from Andrew Ng's notes, except for the extra factor of 2 and the negative sign. This may affect convergence but it is consistent.

In the main loop, the updates are applied using the rule:

$$\theta = \theta - \alpha d\theta$$

Here, the minus sign is used instead of the plus sign in Andrew Ng's notes to match the negative sign in the `update_weights()` function. Again, this is a matter of convention and does not affect the correctness of the algorithm. However, note that the learning rate `learning_rate` is defined as a positive value, so the overall effect is to move the weights in the direction of decreasing loss.

Hence, the update rule implemented in the `update_weights()` function is consistent with the equation from Andrew Ng's notes.

```
sgd_aug_theta = theta
sgd_aug_theta
#sgd means stochastic gradient descent
#the above theta values are very close to the theta values obtained using the psuedo inverse method

array([[ -0.00533449],
       [ 0.00289679],
       [ 0.007446  ]],
```

```
[-0.00392554],  
[-0.00506702],  
[ 0.00041471],  
[-0.0001196 ]])
```

```
Yhat_sgd_aug_val = predict(X_aug_val,sgd_aug_theta)  
Yhat_sgd_aug_val
```

```
[ 0.01066978],  
[ 0.11656721],  
[-0.01698101],  
[ 0.21149119],  
[ 0.1343238 ],  
[ 0.0762704 ],  
[ 0.27992973],  
[-0.06271 ],  
[ 0.1832981 ],  
[ 0.22988329],  
[ 0.22399852],  
[ 0.1805151 ],  
[ 0.26666236],  
[ 0.0239823 ],  
[ 0.26433271],  
[ 0.41109572],  
[ 0.06711274],  
[ 0.18557769],  
[-0.00128196],  
[ 0.05479927],  
[ 0.15577913],  
[ 0.08379492],  
[ 0.03045698],  
[-0.02350147],  
[ 0.35342699],  
[ 0.1944993 ],  
[ 0.04815856],  
[ 0.13571708],  
[ 0.08465262],  
[ 0.12235139],  
[ 0.19596706],  
[ 0.12057771],  
[ 0.02676987],  
[ 0.09476146],  
[ 0.15106045],  
[ 0.12843346],  
[ 0.4443926 ],  
[ 0.14109176],  
[ 0.22810679],  
[ 0.05893804],  
[ 0.39368705],  
[ 0.08127903],  
[ 0.00771641],  
[ 0.11551312],  
[ 0.06366116],  
[ 0.03052614],  
[-0.02273183],  
[ 0.07453254],  
[ 0.2033998 ],  
[ 0.31523917],  
[ 0.36095892],  
[ 0.26904328],  
[ 0.02171719],  
[ 0.37039299],  
[ 0.06362027],  
[ 0.13670413],  
[ 0.06278394],  
[ 0.25450729]])
```

```
E_in = np.mean((Y_train-predict(X_aug_train,sgd_aug_theta))**2)  
E_out = np.mean((Y_val-Yhat_sgd_aug_val)**2)  
print(E_in,E_out)
```

```
0.15558032480472495 0.12746176487814415
```



#### ▼ Q4)

- A. Instead of doing a fixed number of iterations like 1000 above, define a convergence criteria -- it could be for example relative change in dthe
- B. The  $E_{in}$  for SGD is much worse than  $E_{in}$  for pinv method or the sklearn method. It can be improved by
- Setting better initial learning rate
  - Lowering learning rate with iterations

Please experiment with (a) and (b) above to get  $E_{in}$  and  $E_{out}$  for SGD close (very) to  $E_{in}$  and  $E_{out}$  of Pinv method. (15 pts)

Ans B (a) can be implemented by setting the learning rate to 0.0001 but it is too large for this problem and the loss diverges. Any value less than 0.0001 will give us a better learning rate but won't work in this case.

# PART A - we can use the following convergence criteria:

```
#function to compute loss
def compute_loss(X, Y, theta):
    Y_pred = predict(X, theta)
    return np.mean((Y - Y_pred) ** 2)

#function to check for convergence and update theta using gradient descent
learning_rate = 0.00001
#better initial learning rate would be 0.0001 but it is too large for this problem and the loss diverges.
#we can use a smaller learning rate and more iterations to get the same result
theta = init(X_aug_train)

prev_loss = None # keep track of previous loss
tolerance = 1e-5 # set tolerance for convergence

# iterate until convergence or max iterations
for i in range(1000):
    loss = compute_loss(X_aug_train, Y_train, theta) # compute current loss
    # check for convergence
    if prev_loss and abs(prev_loss - loss) < tolerance:
        print("Converged at iteration", i)
        break
    prev_loss = loss #update previous loss
    # update theta
    dtheta = update_weights(X_aug_train, Y_train, theta)
    #print (dtheta)
    theta = theta - learning_rate * dtheta

#the above code is the same as the previous code except that we are checking for convergence

sgd_aug_theta = theta
sgd_aug_theta

Yhat_sgd_aug_val = predict(X_aug_val,sgd_aug_theta)
Yhat_sgd_aug_val

E_in = np.mean((Y_train-predict(X_aug_train,sgd_aug_theta))**2)
E_out = np.mean((Y_val-Yhat_sgd_aug_val)**2)
print(E_in,E_out)

Converged at iteration 464
0.1584946998453554 0.12972081263020044
```

(b) can be implemented by lowering the learning rate with iterations. This will give us a better learning rate. The  $E_{in}$  and  $E_{out}$  for SGD will be close to the  $E_{in}$  and  $E_{out}$  of Pinv method. To do this, we can use the following formula:  $\text{learning\_rate} = \text{learning\_rate} / (1 + \text{decay\_rate} * \text{iteration})$ . This will give us a better learning rate. The  $E_{in}$  and  $E_{out}$  for SGD will be close to the  $E_{in}$  and  $E_{out}$  of Pinv method.

```
#Lowering the learning rate with each iteration
learning_rate = 0.00001
decay_rate = 0.00001
theta = init(X_aug_train)
for i in range(1000):
    dtheta = update_weights(X_aug_train,Y_train, theta)
    #print(dtheta)
    learning_rate = learning_rate/(1+ decay_rate * i)
    theta = theta-learning_rate*dtheta
```

```
sgd_aug_theta = theta
sgd_aug_theta

Yhat_sgd_aug_val = predict(X_aug_val,sgd_aug_theta)
Yhat_sgd_aug_val

E_in = np.mean((Y_train-predict(X_aug_train,sgd_aug_theta))**2)
E_out = np.mean((Y_val-Yhat_sgd_aug_val)**2)
print(E_in,E_out)

0.15926166338996178 0.13006473125306017
```

## ▼ Bonus Q5)

Instead of using the Batch Gradient Descent method above where the updates are done using the entire training data set, use the Stochastic Gradient Descent method, Andrew Ng page 7 top, where updates are done for each training sample.

Repeat Until convergence { for  $i=1$  to  $m$  {  $\theta_j(t+1) = \theta_j(t) + \alpha(y_i - h_\theta(x_i))x_{ij}$  } }

Get similar E\_in and E\_out as above (5 pts)

```
#get same result as above using stochastic gradient descent where updates are done after each training sample
#code is similar to the previous code except that we are using a single training sample to update theta
learning_rate = 0.00001
decay_rate = 0.00001
theta = init(X_aug_train)
for i in range(1000):
    for j in range(X_aug_train.shape[0]):
        dtheta = update_weights(X_aug_train[j:j+1],Y_train[j:j+1], theta)
        #print(dtheta)
        learning_rate = learning_rate/(1+ decay_rate * i)
        theta = theta-learning_rate*dtheta

sgd_aug_theta = theta
sgd_aug_theta

Yhat_sgd_aug_val = predict(X_aug_val,sgd_aug_theta)
Yhat_sgd_aug_val

E_in = np.mean((Y_train-predict(X_aug_train,sgd_aug_theta))**2)
E_out = np.mean((Y_val-Yhat_sgd_aug_val)**2)
print(E_in,E_out)

0.14752070233225242 0.1218631050602129
```

## ▼ Q6)

Rewrite above Stochastic Batch Gradient Descent to work over batches. Let there be  $b$  batches each of size  $\leq k$

Repeat Until convergence { for  $i=1$  to  $b$  {  $\theta_j(t+1) = \theta_j(t) + \alpha \sum_{i=1}^k (y_i - h_\theta(x_i))x_{ij}$  } }

Note that if say your training sample is for example 910 samples, there will be 9 batches of 100 samples and one batch 10 samples. So there will be 10 batches that will constitute an "epoch". And there will be multiple iterations over the epoch to get good result. (10 pts)

```
#get same result as above using stochastic gradient descent where updates are done after batch of training samples
#batch size is 10

b = 10
learning_rate = 0.00001
decay_rate = 0.00001
theta = init(X_aug_train)
for i in range(1000):
    for j in range(0,X_aug_train.shape[0],b):
        dtheta = update_weights(X_aug_train[j:j+1],Y_train[j:j+1], theta)
        #print(dtheta)
        learning_rate = learning_rate/(1+ decay_rate * i)
        theta = theta-learning_rate*dtheta

sgd_aug_theta = theta
sgd_aug_theta

Yhat_sgd_aug_val = predict(X_aug_val,sgd_aug_theta)
Yhat_sgd_aug_val

E_in = np.mean((Y_train-predict(X_aug_train,sgd_aug_theta))**2)
```

```
E_out = np.mean((Y_val-Yhat_sgd_aug_val)**2)
print(E_in,E_out)
```

```
0.1520732942800172 0.13021867240335358
```

## ▼ Q7)

To get better results one can regularize the cost function. Original cost function is

$$J(\theta) = \sum_{i=0}^m (y_i - \theta x_i)^2$$

the regularized cost function is

$$J(\theta) = \sum_{i=0}^m (y_i - \theta x_i)^2 + \lambda \|\theta\|_2$$

where  $\lambda$  is a regularization constant (set by you) and  $\|\cdot\|_2$  is the  $L_2$  norm. For this cost function find  $\theta$  and then do experiment Q4 B again to report best  $E_{in}$  and  $E_{out}$  results (20 pts)

```
#defining a regularized loss function
def compute_loss_reg(X, Y, theta, reg):
    Y_pred = predict(X, theta)
    return np.mean((Y - Y_pred) ** 2) + reg * np.mean(theta ** 2)

#regularized update weights function
def update_weights_reg(X, Y, theta, reg):
    Y_pred = predict(X, theta)
    dtheta = -2 * np.mean((Y - Y_pred) * X, axis=0) + 2 * reg * theta
    return dtheta

#regularized stochastic gradient descent
learning_rate = 0.00001
decay_rate = 0.00001
reg = 0.00001
theta = init(X_aug_train)
for i in range(1000):
    for j in range(0,X_aug_train.shape[0],b):
        dtheta = update_weights_reg(X_aug_train[j:j+1],Y_train[j:j+1], theta, reg)
        #print(dtheta)
        learning_rate = learning_rate/(1+ decay_rate * i)
        theta = theta-learning_rate*dtheta

sgd_aug_theta = theta
sgd_aug_theta

Yhat_sgd_aug_val = predict(X_aug_val,sgd_aug_theta)
Yhat_sgd_aug_val

E_in = np.mean((Y_train-predict(X_aug_train,sgd_aug_theta))**2)
E_out = np.mean((Y_val-Yhat_sgd_aug_val)**2)
print(E_in,E_out)

#here we are using the regularized stochastic gradient descent to find the best regularization parameter
#reg = lambda in the regularized loss function above

0.17614519401456386 0.13925680769102544
```