Biswas_Sayan_HW2

October 9, 2019

[1]:
```
import pandas as pd
import numpy as np
from sklearn import metrics
from sklearn import preprocessing
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import math
```

1 Problem 1

1.1 1a

[2]:
```
cols = list(pd.read_csv("train.csv", nrows =1))
train = pd.read_csv("train.csv", usecols =[i for i in cols if i not in␣
 ↪["id","date","zipcode"]])
train = train.drop(columns=['Unnamed: 0'])

X_train = train.iloc[:,1:]
Y_train = train.iloc[:,0:1]

model_a = LinearRegression().fit(X_train, Y_train)
coeff = pd.DataFrame(model_a.coef_.T,X_train.columns,columns=['Coefficients'])
print("Intercept : ",model_a.intercept_[0])
coeff
```

```
Intercept :  -23088898.60853133
```

[2]:

	Coefficients
bedrooms	-14704.280497
bathrooms	25687.783987
sqft_living	83.084210
sqft_lot	0.375930
floors	15555.580988
waterfront	715535.170469
view	63027.898001
condition	18816.402756

```
grade            79534.602722
sqft_above          42.010495
sqft_basement       41.073715
yr_built         -2400.669330
yr_renovated        43.682942
lat             553505.032276
long             -7424.027121
sqft_living15       68.015792
sqft_lot15          -0.515528
```

[3]:

```
Y_pred = model_a.predict(X_train)
print('Mean Squared Error for train data using model_a:', metrics.
 ↪mean_squared_error(Y_train, Y_pred))
```

```
Mean Squared Error for train data using model_a: 31486167775.7949
```

1.2 1b

[4]:

```
ss_scaler = preprocessing.StandardScaler()
train_col = train.columns
scaled_data = pd.DataFrame(ss_scaler.fit_transform(train.values),columns =␣
 ↪train_col)
data = scaled_data.copy()
X_scaled = data.iloc[:,1:]
Y_scaled = data[["price"]]
# model_b = LinearRegression().fit(X_scaled, Y_scaled)

model_b = LinearRegression().fit(X_scaled, Y_scaled)
coeff = pd.DataFrame(model_b.coef_.T,X_scaled.columns,columns=['Coefficients'])
print("Intercept : ",model_b.intercept_[0])
coeff
```

```
Intercept :  3.086029186623574e-15
```

[4]:

	Coefficients
bedrooms	-0.036903
bathrooms	0.054602
sqft_living	0.167243
sqft_lot	0.032070
floors	0.023706
waterfront	0.187856
view	0.142050
condition	0.038207
grade	0.271814
sqft_above	0.142315
sqft_basement	0.079975
yr_built	-0.199350

```
yr_renovated       0.050900
lat                0.230980
long              -0.003051
sqft_living15      0.134321
sqft_lot15        -0.038106
```

[5]:
```
Y_pred_scaled = model_b.predict(X_scaled)
print('Mean Squared Error for training data using model_b:', metrics.
 ↪mean_squared_error(Y_scaled, Y_pred_scaled))
```

```
Mean Squared Error for training data using model_b: 0.2734665681293983
```

1.3 1c

[6]:
```
cols = list(pd.read_csv("test.csv", nrows =1))
test = pd.read_csv("test.csv", usecols =[i for i in cols if i not in␣
 ↪["id","date","zipcode"]])
test = test.iloc[:,1:]
X_test = test.iloc[:,1:]
Y_test = test.iloc[:,0:1]
Y_pred_test = model_a.predict(X_test)
print('Mean Squared Error for test data using model_a:', metrics.
 ↪mean_squared_error(Y_test, Y_pred_test))
```

```
Mean Squared Error for test data using model_a: 57628154705.675156
```

[7]:
```
test_col = test.columns
scaled_data_test = pd.DataFrame(ss_scaler.transform(test.
 ↪values),columns=test_col)
X_test_scaled = scaled_data_test.iloc[:,1:].values
Y_test_scaled = scaled_data_test.iloc[:,0:1].values
Y_pred_test_scaled = model_b.predict(X_test_scaled)
print('Mean Squared Error for test data using model_b:', metrics.
 ↪mean_squared_error(Y_test_scaled, Y_pred_test_scaled))
```

```
Mean Squared Error for test data using model_b: 0.5005173639170148
```

1.4 1d

The coefficients obtained after learning from the unscaled data cannot be compared as they are in different units of measurements respectively. However after feature scaling of the training data, the coefficients can be compared to one another as the features where brought to the same scale before training. The MSE obtained from the unscaled data is a large number and not very intuitive compared to the MSE obtained from the test data to be compared. The features grade, latitude, view, waterfront, sqft_living,sqft_living15,sqft_above and yr_built mostly contribute to the linear regression model based on the coefficients.

[8]:
```python
r_squared= metrics.r2_score(Y_test_scaled, Y_pred_test_scaled)
print('coefficient of determination:', r_squared)
```

```
coefficient of determination: 0.6543560876120955
```

Based on the R-Squared error, 65.4% of the variation in y can be explained by the dependence on features using the regression model

The model error is 0.5005173639170148.

2 Problem 2

2.1 2 a,b

[9]:
```python
def ClosedFormSoln(X, Y):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    X=X.values
    Y=Y.values
    ## calculating the closed form solution to get parameters theta
    X_transpose = np.transpose(X)
    # calculating the dot product
    X_transpose_dotp_x = X_transpose.dot(X)
    # Calculating the inverse
    temp_1 = np.linalg.pinv(X_transpose_dotp_x)

    ##Calculating the second half i.e. (X transpose Y)
    temp_2 = X_transpose.dot(Y)

    ##Calculating theta
    theta = temp_1.dot(temp_2)
    return(theta)

def predict_ClosedFormSoln(theta, X):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    X=X.values
    Y_pred = X.dot(theta)
    return Y_pred
```

[10]:
```python
#Initializing data
train = scaled_data.copy()
test = scaled_data_test.copy()
# Y values
Y_train = train[["price"]]
Y_test = test[["price"]]
```

```
X_train = train[["sqft_living"]]
X_test = test[["sqft_living"]]
```

[11]:
```
# X_test can be a single sample or multiple sample (X_test =␣
↪test[["sqft_living"]])
X_test = X_test.iloc[0:1,:]
```

[12]:
```
theta_single_dim = ClosedFormSoln(X_train,Y_train)
Y_pred = predict_ClosedFormSoln(theta_single_dim,X_test)
print('Response for a new single-dimensional data point: ', Y_pred[0])
```

```
Response for a new single-dimensional data point:  [-0.17565766]
```

[13]:
```
#Initializing data
train = scaled_data.copy()
test = scaled_data_test.copy()
# Y values
Y_train = train[["price"]]
Y_test = test[["price"]]
X_train = train.iloc[:,1:]
X_test = test.iloc[:,1:]
```

[14]:
```
# X_test can be a single sample here in the code snippet it takes the first row␣
↪of the test data
# or multiple rows using (X_test = test.iloc[:,1:])
X_test = X_test.iloc[0:1,:]
```

[15]:
```
theta_multi_dim = ClosedFormSoln(X_train,Y_train)
Y_pred = predict_ClosedFormSoln(theta_multi_dim,X_test)
print('Response for a new multi-dimensional data point: ', Y_pred[0])
```

```
Response for a new multi-dimensional data point:  [0.45660638]
```

2.2 2c

Since the closed form gives the global optimum hence the parameters obtained from the closed form solution are same as those obtained from the linear model implemented using the packages. This is the reason the MSE values obtained using the closed form implementation are similar to those obtained using the packages as seen from the output below.

[16]:
```
#Initializing data
train = scaled_data.copy()
test = scaled_data_test.copy()
# Y values
Y_train = train[["price"]]
Y_test = test[["price"]]
X_train = train.iloc[:,1:]
```

```
X_test = test.iloc[:,1:]
```

[17]:
```
Y_pred_train = predict_ClosedFormSoln(theta_multi_dim,X_train)
Y_pred_test = predict_ClosedFormSoln(theta_multi_dim,X_test)
print('Mean Squared Error for training data using closed form solution:',␣
 ↪metrics.mean_squared_error(Y_train, Y_pred_train))
print('Mean Squared Error for test data using closed form solution:', metrics.
 ↪mean_squared_error(Y_test, Y_pred_test))
```

```
Mean Squared Error for training data using closed form solution:
0.2734665681293983
Mean Squared Error for test data using closed form solution: 0.5005173639170145
```

3 Problem 3

3.1 3a

[18]:
```
# Compute Cost function is implemented to check if the cost is decreasing with␣
 ↪iterations
def ComputeCost(X,Y,theta):
    N = Y.shape[0]
    temp = X.dot(theta) - Y
    cost = (1/N)*np.transpose(temp).dot(temp)
    return cost

# Method to compute the theta values using gradient descent
def gradient_descent(X, Y, alpha, num_iters):
    iters = num_iters
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    # d is the number of features
    d = X_train.shape[1]
    # Initializing theta with zeros
    theta = np.zeros((d,1))
    X=X.values
    Y=Y.values
    diff_cost = 0
    N = Y.shape[0]
    theta_old = theta
    #while diff_cost > 0.000001 or num_iters > 0
    while num_iters > 0:
        temp = np.transpose(X.dot(theta_old) - Y).dot(X)
        theta_new = theta_old - alpha * (2/N) * np.transpose(temp)
        # checking for convergence
        delta_theta = theta_new - theta_old
        delta = np.sqrt(np.transpose(delta_theta).dot(delta_theta))
```

```
        if delta < 0.00001 and iters != num_iters :
            print("Gradient descent converged at iteration", iters-num_iters)
            break
        num_iters = num_iters - 1
        theta_old = theta_new
        #old_cost = ComputeCost(X,Y,theta_old)
        #new_cost = ComputeCost(X,Y,theta_new)
        #diff_cost = old_cost - new_cost
    return theta_new

# predict the y values using theta and X
def predict_GradientDescent(theta, X):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    X = X.values
    Y_pred = X.dot(theta)
    return Y_pred
```

3.2 3b

[19]:
```
train = scaled_data.copy()
test = scaled_data_test.copy()
# Y values
Y_train = train[["price"]]
Y_test = test[["price"]]
X_train = train.iloc[:,1:]
X_test = test.iloc[:,1:]
```

[20]:
```
alpha = [0.01, 0.03, 0.1]
num_iters = [10, 50, 100]

for i in alpha:
    for j in num_iters :
        theta = gradient_descent(X_train, Y_train, i, j)
        print('theta when alpha = ' + str(i) + ' and num_iters = ' + str(j) + ':
 ↪\n', theta)
        print("\n")
        Y_pred_train = predict_GradientDescent(theta, X_train)
        Y_pred_test = predict_GradientDescent(theta, X_test)
        print('MSE for training data using Gradient descent when alpha = ' +␣
 ↪str(i) +
              ' and num_iters = ' + str(j) + ':', metrics.
 ↪mean_squared_error(Y_train, Y_pred_train))
        print('MSE for testing data using Gradient descent when alpha = ' +␣
 ↪str(i) +
```

```
            ' and num_iters = '+ str(j) + ':', metrics.
↪mean_squared_error(Y_test, Y_pred_test))
        print("\n")
```

```
theta when alpha = 0.01 and num_iters = 10:
 [[ 8.24229573e-17]
 [ 3.51546174e-02]
 [ 5.97764565e-02]
 [ 9.46499119e-02]
 [ 1.58577098e-02]
 [ 2.80724205e-02]
 [ 5.14216548e-02]
 [ 6.74806965e-02]
 [ 1.46048174e-02]
 [ 8.81429767e-02]
 [ 7.58100530e-02]
 [ 5.35522376e-02]
 [-1.19508461e-02]
 [ 2.46116923e-02]
 [ 6.15128542e-02]
 [-4.24986932e-03]
 [ 8.66260205e-02]
 [ 1.73904498e-02]]


MSE for training data using Gradient descent when alpha = 0.01 and num_iters =
10: 0.4769826562643797
MSE for testing data using Gradient descent when alpha = 0.01 and num_iters =
10: 0.7899805878552555


theta when alpha = 0.01 and num_iters = 50:
 [[ 6.05232531e-16]
 [ 1.76980518e-02]
 [ 7.02037279e-02]
 [ 1.61104777e-01]
 [ 1.07199113e-02]
 [ 3.27773917e-02]
 [ 1.35936441e-01]
 [ 1.46319888e-01]
 [ 4.29116775e-02]
 [ 1.66822006e-01]
 [ 1.25277756e-01]
 [ 9.77393699e-02]
 [-9.43446724e-02]
 [ 6.27487154e-02]
```

```
 [ 1.77221329e-01]
 [-3.88336062e-02]
 [ 1.51099990e-01]
 [ 7.90555822e-03]]
```

```
MSE for training data using Gradient descent when alpha = 0.01 and num_iters =
50: 0.2935883463907345
MSE for testing data using Gradient descent when alpha = 0.01 and num_iters =
50: 0.5343327522759017
```

```
theta when alpha = 0.01 and num_iters = 100:
 [[ 1.23490662e-15]
 [-1.07772411e-02]
 [ 5.67070436e-02]
 [ 1.67851095e-01]
 [ 7.93841518e-03]
 [ 3.07915991e-02]
 [ 1.65358934e-01]
 [ 1.54682331e-01]
 [ 4.72356963e-02]
 [ 1.95563835e-01]
 [ 1.37140433e-01]
 [ 9.02383856e-02]
 [-1.35234073e-01]
 [ 6.71310263e-02]
 [ 2.21279091e-01]
 [-3.74686822e-02]
 [ 1.61148092e-01]
 [-3.15262960e-03]]
```

```
MSE for training data using Gradient descent when alpha = 0.01 and num_iters =
100: 0.27842422313332665
MSE for testing data using Gradient descent when alpha = 0.01 and num_iters =
100: 0.5124065597546549
```

```
theta when alpha = 0.03 and num_iters = 10:
 [[ 3.59918761e-16]
 [ 3.44783187e-02]
 [ 7.77279035e-02]
 [ 1.51030380e-01]
 [ 1.49096249e-02]
 [ 3.45824114e-02]
 [ 1.09451543e-01]
 [ 1.28344010e-01]
```

```
 [ 3.53335842e-02]
 [ 1.47384928e-01]
 [ 1.16499005e-01]
 [ 9.32828025e-02]
 [-6.15572702e-02]
 [ 5.28254858e-02]
 [ 1.38219844e-01]
 [-2.88618577e-02]
 [ 1.39434856e-01]
 [ 1.51205380e-02]]


MSE for training data using Gradient descent when alpha = 0.03 and num_iters =
10: 0.32025061817365497
MSE for testing data using Gradient descent when alpha = 0.03 and num_iters =
10: 0.5706630290510314


theta when alpha = 0.03 and num_iters = 50:
 [[ 1.81494153e-15]
 [-2.42677316e-02]
 [ 5.09297569e-02]
 [ 1.69308284e-01]
 [ 8.93562133e-03]
 [ 2.85160307e-02]
 [ 1.75952382e-01]
 [ 1.50939455e-01]
 [ 4.68466787e-02]
 [ 2.13358145e-01]
 [ 1.42588380e-01]
 [ 8.35618907e-02]
 [-1.54222591e-01]
 [ 6.42322034e-02]
 [ 2.34214776e-01]
 [-2.88323918e-02]
 [ 1.62387196e-01]
 [-1.01317941e-02]]


MSE for training data using Gradient descent when alpha = 0.03 and num_iters =
50: 0.27560896108344324
MSE for testing data using Gradient descent when alpha = 0.03 and num_iters =
50: 0.5074745831005474


theta when alpha = 0.03 and num_iters = 100:
 [[ 2.66659361e-15]
 [-3.47525840e-02]
```

```
 [ 4.93272743e-02]
 [ 1.69715918e-01]
 [ 1.56037395e-02]
 [ 2.41618518e-02]
 [ 1.84600950e-01]
 [ 1.43066441e-01]
 [ 4.33028607e-02]
 [ 2.41936788e-01]
 [ 1.45193915e-01]
 [ 7.97991240e-02]
 [-1.79016692e-01]
 [ 5.72099614e-02]
 [ 2.35964039e-01]
 [-1.42836817e-02]
 [ 1.54438690e-01]
 [-2.21623928e-02]]


MSE for training data using Gradient descent when alpha = 0.03 and num_iters =
100: 0.27395671526270826
MSE for testing data using Gradient descent when alpha = 0.03 and num_iters =
100: 0.5030789048604056


theta when alpha = 0.1 and num_iters = 10:
 [[ 1.24420474e-15]
 [-1.36910568e-02]
 [ 5.52088472e-02]
 [ 1.68353541e-01]
 [ 7.45623778e-03]
 [ 3.10991810e-02]
 [ 1.68132153e-01]
 [ 1.55868304e-01]
 [ 4.76371951e-02]
 [ 1.97351136e-01]
 [ 1.38347012e-01]
 [ 8.91136198e-02]
 [-1.38420055e-01]
 [ 6.78915154e-02]
 [ 2.26265130e-01]
 [-3.78793254e-02]
 [ 1.62485721e-01]
 [-3.72317254e-03]]


MSE for training data using Gradient descent when alpha = 0.1 and num_iters =
10: 0.2777784544544066
MSE for testing data using Gradient descent when alpha = 0.1 and num_iters = 10:
```

```
0.5113613440792656


theta when alpha = 0.1 and num_iters = 50:
 [[ 2.94542168e-15]
 [-3.67123703e-02]
 [ 5.19311794e-02]
 [ 1.68537088e-01]
 [ 2.31442397e-02]
 [ 2.35943958e-02]
 [ 1.86936278e-01]
 [ 1.41325823e-01]
 [ 4.01410188e-02]
 [ 2.59030114e-01]
 [ 1.43764344e-01]
 [ 7.99826561e-02]
 [-1.91538284e-01]
 [ 5.33598604e-02]
 [ 2.32971158e-01]
 [-7.57832894e-03]
 [ 1.44321559e-01]
 [-2.98970970e-02]]


MSE for training data using Gradient descent when alpha = 0.1 and num_iters =
50: 0.27356291763134366
MSE for testing data using Gradient descent when alpha = 0.1 and num_iters = 50:
0.5013370393181915


theta when alpha = 0.1 and num_iters = 100:
 [[ 3.07953663e-15]
 [-3.68947451e-02]
 [ 5.42975971e-02]
 [ 1.67304747e-01]
 [ 3.02686805e-02]
 [ 2.37564001e-02]
 [ 1.87723494e-01]
 [ 1.41848046e-01]
 [ 3.83994393e-02]
 [ 2.70127986e-01]
 [ 1.42361073e-01]
 [ 8.00147289e-02]
 [-1.98462669e-01]
 [ 5.11892697e-02]
 [ 2.31198797e-01]
 [-3.65404153e-03]
 [ 1.35919784e-01]
```

```
 [-3.63918536e-02]]


MSE for training data using Gradient descent when alpha = 0.1 and num_iters =
100: 0.27346886118648905
MSE for testing data using Gradient descent when alpha = 0.1 and num_iters =
100: 0.5006111869631489
```

3.3 3c

The MSE value for the training and the testing data using Gradient Descent for Alpha = 0.1 and iterations = 100 with theta being initialized with all zeroes; is almost similar to those obtained with the package. The more the number of iterations, the closer the MSE is to the one obtained using the package. We can observe that even with smaller learning rate, the algorithm converges if provided with sufficient number of iterations. The MSE value is small for a bigger learning rate compared to the small learning rates for the same number of iterations. The objective decreases for the selected learning rates with each iterations however if the learning rate is selected as 0.2 the cost increases with each iteration and hence cannot be selected as an alpha value for gradient descent. Using the above implementation, the algorithm is converging with the number of iterations reaching the maximum number of iterations.

3.4 3d

[21]:
```
# Compute Cost function is implemented to check if the cost is decreasing with␣
↪iterations
def ComputeCost(X,Y,theta):
    N = Y.shape[0]
    temp = X.dot(theta) - Y
    cost = (1/N)*np.transpose(temp).dot(temp)
    return cost

# Method to compute the theta values using gradient descent
def gradient_descent_line_search(X, Y, alpha_max, step, e, T):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    # d is the number of features
    d = X_train.shape[1]
    # Initializing theta with zeros
    theta = np.zeros((d,1))

    X=X.values
    Y=Y.values
    N = Y.shape[0]
    delta = 1
    alpha = alpha_max
    backtrack = T
    while(delta > 0.0001):
        T = backtrack
        theta_try = theta
        while(T>0):
            temp = np.transpose(X.dot(theta) - Y).dot(X)
            theta_try = theta - alpha * (2/N) * np.transpose(temp)
            if abs((ComputeCost(X,Y, theta) - ComputeCost(X,Y, theta_try))) > e:
                theta = theta_try
                #break
            else:
                alpha = step * alpha
            T = T-1
        delta_theta = theta_try - theta
        delta = np.sqrt(np.transpose(delta_theta).dot(delta_theta))
    return theta


# predict the y values using theta and X
def predict_GradientDescent(theta, X):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
```

```
        X.insert(0, "x0", x0, True)
    X = X.values
    Y_pred = X.dot(theta)
    return Y_pred
```

[22]:
```
train = scaled_data.copy()
test = scaled_data_test.copy()
# Y values
Y_train = train[["price"]]
Y_test = test[["price"]]
X_train = train.iloc[:,1:]
X_test = test.iloc[:,1:]
```

[23]:
```
alpha = [0.01, 0.03, 0.1]
backtrack = [10, 50, 100]

for i in alpha:
    for j in backtrack :
        theta = gradient_descent_line_search(X_train, Y_train, i, 0.5, 0.0001, j)
        #print('theta when alpha = ' + str(i) + ' and num_iters = ' + str(j) + ':
 ↪\n', theta)
        #print("\n")
        Y_pred_train = predict_GradientDescent(theta, X_train)
        Y_pred_test = predict_GradientDescent(theta, X_test)
        print('MSE for training data using Gradient descent when alpha = ' +␣
 ↪str(i) +
              ' and backtrack = ' + str(j) + ':', metrics.
 ↪mean_squared_error(Y_train, Y_pred_train))
        print('MSE for testing data using Gradient descent when alpha = ' +␣
 ↪str(i) +
              ' and backtrack = '+ str(j) + ':', metrics.
 ↪mean_squared_error(Y_test, Y_pred_test))
        print("\n")
```

```
MSE for training data using Gradient descent when alpha = 0.01 and backtrack =
10: 0.4769826562643797
MSE for testing data using Gradient descent when alpha = 0.01 and backtrack =
10: 0.7899805878552555


MSE for training data using Gradient descent when alpha = 0.01 and backtrack =
50: 0.2935883463907345
MSE for testing data using Gradient descent when alpha = 0.01 and backtrack =
50: 0.5343327522759017


MSE for training data using Gradient descent when alpha = 0.01 and backtrack =
```

```
100: 0.27842422313332665
MSE for testing data using Gradient descent when alpha = 0.01 and backtrack =
100: 0.5124065597546549


MSE for training data using Gradient descent when alpha = 0.03 and backtrack =
10: 0.32025061817365497
MSE for testing data using Gradient descent when alpha = 0.03 and backtrack =
10: 0.5706630290510314


MSE for training data using Gradient descent when alpha = 0.03 and backtrack =
50: 0.27588674681517505
MSE for testing data using Gradient descent when alpha = 0.03 and backtrack =
50: 0.5080309707498859


MSE for training data using Gradient descent when alpha = 0.03 and backtrack =
100: 0.27588674681517505
MSE for testing data using Gradient descent when alpha = 0.03 and backtrack =
100: 0.5080309707498859


MSE for training data using Gradient descent when alpha = 0.1 and backtrack =
10: 0.2777784544544066
MSE for testing data using Gradient descent when alpha = 0.1 and backtrack = 10:
0.5113613440792656


MSE for training data using Gradient descent when alpha = 0.1 and backtrack =
50: 0.2745099563397425
MSE for testing data using Gradient descent when alpha = 0.1 and backtrack = 50:
0.5048703004874806


MSE for training data using Gradient descent when alpha = 0.1 and backtrack =
100: 0.2745099563397425
MSE for testing data using Gradient descent when alpha = 0.1 and backtrack =
100: 0.5048703004874806
```

The values of MSE in the training and testing data using gradient descent using line search is almost similar to those obtained using Gradient descent.

4 Problem 4

Problem 4

a) $J(\theta) = \frac{1}{2}\sum_{i=1}^{N}\left(h_\theta(x_i) - y_i\right)^2 + \frac{1}{2}\cdot\lambda\sum_{j=1}^{d}\theta_j^2$

$J(\theta)$ can also be written as, the below equation is vector Notation.

$$J(\theta) = \frac{1}{2}\|X\theta - y\|^2 + \frac{\lambda}{2}\|\theta\|^2$$

To find, $\min_{\theta} J(\theta)$,

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{2}\times 2\,(X\theta - Y)^T.X + \frac{\lambda}{2}\times 2\,\theta^T$$

$$= (X\theta - Y)^T.X + \lambda\theta^T$$

to find the minimum,

Equating $\frac{\partial J(\theta)}{\partial \theta} = 0$ to find the minimum.

$$\Rightarrow (X\theta - Y)^T X + \lambda\theta^T = 0$$

Transposing on both the sides, we get,

$$\Rightarrow \left((X\theta - Y)^T X + \lambda\theta^T\right)^T = 0^T$$

$$\Rightarrow \left((X\theta - Y)^T X\right)^T + (\lambda\theta^T)^T = 0$$

$$\Rightarrow X^T(X\theta - Y) + \lambda\theta = 0$$

$$\Rightarrow X^T X\theta - X^T Y + \lambda\theta = 0$$

$$\Rightarrow X^T X\theta + \lambda\theta - X^T Y = 0$$

$$\Rightarrow (X^T X + \lambda I)\theta - X^T Y = 0 \qquad [\because \lambda I = \lambda]$$

$$\Rightarrow \theta = (X^T X + \lambda I)^{-1} X^T Y$$

Scanned by CamScanner

4.1 4b

[24]:
```
def ClosedForm_RidgeRegression(X, Y, lambda_r):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    X=X.values
    Y=Y.values
    ## calculating the closed form solution to get parameters theta
    X_transpose = np.transpose(X)
    # calculating the dot product
    X_transpose_dotp_x = X_transpose.dot(X)

    # Matrix for Regularization term
    L = np.identity(X.shape[1])
    #No regularization/penalizing on the theta0
    L[0][0]=0
    # Calculating the inverse
    temp_1 = np.linalg.pinv(X_transpose_dotp_x + (lambda_r * L))

    ##Calculating the second half i.e. (X transpose Y)
    temp_2 = X_transpose.dot(Y)

    ##Calculating theta
    theta = temp_1.dot(temp_2)
    return(theta)

def predict(theta, X):
    if 'x0' not in X.columns:
        x0 = np.ones((X.shape[0], 1), dtype=int)
        X.insert(0, "x0", x0, True)
    Y_pred = X.dot(theta)
    return Y_pred
```

[25]:
```
#Initializing data
train = scaled_data.copy()
test = scaled_data_test.copy()
# Y values
Y_train = train[["price"]]
Y_test = test[["price"]]
X_train = train.iloc[:,1:]
X_test = test.iloc[:,1:]
```

[26]:
```
#lambda_list = [0,1,2,5,10,15,20,30,50,75,100,500,1000,2000,3000,5000]
#lambda_list = [0,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]
lambda_list = []
MSE_train = []
```

```
#for i in np.arange(1,100,1):
for i in range(1,101,1):
#for i in lambda_list:
    lambda_list.append(i)
    theta = ClosedForm_RidgeRegression(X_train,Y_train,i)
    #print(theta)
    Y_pred = predict(theta,X_train)
    print('Mean Squared Error for training data using closed form solution with␣
 ↪lambda= '+ str(i) + ': ',
          metrics.mean_squared_error(Y_train, Y_pred))
    #print("\n")
    MSE_train.append(metrics.mean_squared_error(Y_train, Y_pred))

min_lambda = lambda_list[MSE_train.index(min(MSE_train))]
print("\n")
print("Minimum MSE value for the training data = ", min(MSE_train), "obtained␣
 ↪for lambda =", min_lambda)
plt.plot(lambda_list,MSE_train)
plt.title("Lambda v/s MSE on the training data ")
plt.xlabel("Lambda")
plt.ylabel("MSE")
plt.show()
```

```
Mean Squared Error for training data using closed form solution with lambda= 1:
0.2734668788542625
Mean Squared Error for training data using closed form solution with lambda= 2:
0.27346780421401556
Mean Squared Error for training data using closed form solution with lambda= 3:
0.2734693341886705
Mean Squared Error for training data using closed form solution with lambda= 4:
0.2734714590206986
Mean Squared Error for training data using closed form solution with lambda= 5:
0.2734741692062795
Mean Squared Error for training data using closed form solution with lambda= 6:
0.27347745548689273
Mean Squared Error for training data using closed form solution with lambda= 7:
0.2734813088412359
Mean Squared Error for training data using closed form solution with lambda= 8:
0.2734857204774564
Mean Squared Error for training data using closed form solution with lambda= 9:
0.2734906818256808
Mean Squared Error for training data using closed form solution with lambda= 10:
0.2734961845308304
Mean Squared Error for training data using closed form solution with lambda= 11:
0.2735022204457091
Mean Squared Error for training data using closed form solution with lambda= 12:
0.2735087816243536
```

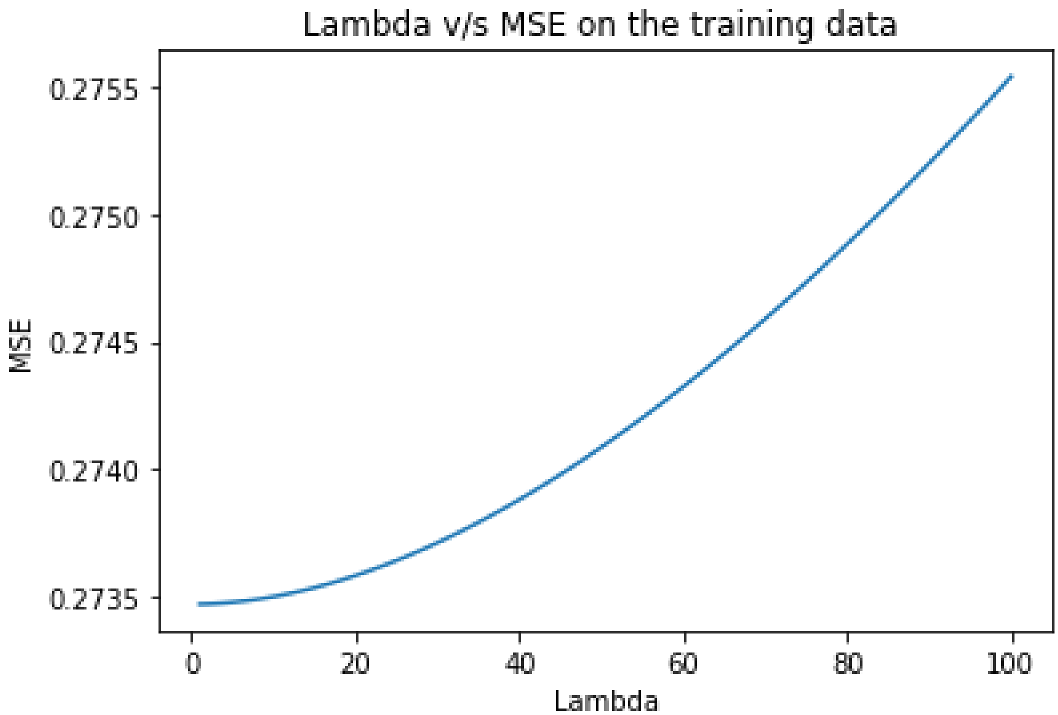
```
Mean Squared Error for training data using closed form solution with lambda= 13:
0.27351586031563274
Mean Squared Error for training data using closed form solution with lambda= 14:
0.2735234489570864
Mean Squared Error for training data using closed form solution with lambda= 15:
0.27353154016899406
Mean Squared Error for training data using closed form solution with lambda= 16:
0.27354012674866146
Mean Squared Error for training data using closed form solution with lambda= 17:
0.27354920166491864
Mean Squared Error for training data using closed form solution with lambda= 18:
0.2735587580528187
Mean Squared Error for training data using closed form solution with lambda= 19:
0.27356878920852923
Mean Squared Error for training data using closed form solution with lambda= 20:
0.27357928858440955
Mean Squared Error for training data using closed form solution with lambda= 21:
0.27359024978426405
Mean Squared Error for training data using closed form solution with lambda= 22:
0.2736016665587666
Mean Squared Error for training data using closed form solution with lambda= 23:
0.27361353280104744
Mean Squared Error for training data using closed form solution with lambda= 24:
0.2736258425424361
Mean Squared Error for training data using closed form solution with lambda= 25:
0.27363858994835594
Mean Squared Error for training data using closed form solution with lambda= 26:
0.2736517693143609
Mean Squared Error for training data using closed form solution with lambda= 27:
0.2736653750623118
Mean Squared Error for training data using closed form solution with lambda= 28:
0.27367940173668504
Mean Squared Error for training data using closed form solution with lambda= 29:
0.2736938440010089
Mean Squared Error for training data using closed form solution with lambda= 30:
0.27370869663442243
Mean Squared Error for training data using closed form solution with lambda= 31:
0.2737239545283518
Mean Squared Error for training data using closed form solution with lambda= 32:
0.27373961268330016
Mean Squared Error for training data using closed form solution with lambda= 33:
0.2737556662057459
Mean Squared Error for training data using closed form solution with lambda= 34:
0.2737721103051456
Mean Squared Error for training data using closed form solution with lambda= 35:
0.2737889402910377
Mean Squared Error for training data using closed form solution with lambda= 36:
0.2738061515702425
```

```
Mean Squared Error for training data using closed form solution with lambda= 37:
0.2738237396441549
Mean Squared Error for training data using closed form solution with lambda= 38:
0.27384170010612796
Mean Squared Error for training data using closed form solution with lambda= 39:
0.2738600286389406
Mean Squared Error for training data using closed form solution with lambda= 40:
0.2738787210123495
Mean Squared Error for training data using closed form solution with lambda= 41:
0.273897773080721
Mean Squared Error for training data using closed form solution with lambda= 42:
0.2739171807807383
Mean Squared Error for training data using closed form solution with lambda= 43:
0.273936940129184
Mean Squared Error for training data using closed form solution with lambda= 44:
0.27395704722079367
Mean Squared Error for training data using closed form solution with lambda= 45:
0.27397749822617773
Mean Squared Error for training data using closed form solution with lambda= 46:
0.2739982893898091
Mean Squared Error for training data using closed form solution with lambda= 47:
0.2740194170280754
Mean Squared Error for training data using closed form solution with lambda= 48:
0.27404087752739237
Mean Squared Error for training data using closed form solution with lambda= 49:
0.27406266734237567
Mean Squared Error for training data using closed form solution with lambda= 50:
0.27408478299407113
Mean Squared Error for training data using closed form solution with lambda= 51:
0.2741072210682389
Mean Squared Error for training data using closed form solution with lambda= 52:
0.27412997821369106
Mean Squared Error for training data using closed form solution with lambda= 53:
0.27415305114068067
Mean Squared Error for training data using closed form solution with lambda= 54:
0.27417643661934016
Mean Squared Error for training data using closed form solution with lambda= 55:
0.2742001314781671
Mean Squared Error for training data using closed form solution with lambda= 56:
0.2742241326025562
Mean Squared Error for training data using closed form solution with lambda= 57:
0.27424843693337575
Mean Squared Error for training data using closed form solution with lambda= 58:
0.27427304146558695
Mean Squared Error for training data using closed form solution with lambda= 59:
0.2742979432469046
Mean Squared Error for training data using closed form solution with lambda= 60:
0.2743231393764978
```

```
Mean Squared Error for training data using closed form solution with lambda= 61:
0.2743486270037289
Mean Squared Error for training data using closed form solution with lambda= 62:
0.2743744033269306
Mean Squared Error for training data using closed form solution with lambda= 63:
0.2744004655922183
Mean Squared Error for training data using closed form solution with lambda= 64:
0.2744268110923376
Mean Squared Error for training data using closed form solution with lambda= 65:
0.2744534371655453
Mean Squared Error for training data using closed form solution with lambda= 66:
0.27448034119452336
Mean Squared Error for training data using closed form solution with lambda= 67:
0.27450752060532385
Mean Squared Error for training data using closed form solution with lambda= 68:
0.274534972866344
Mean Squared Error for training data using closed form solution with lambda= 69:
0.27456269548733175
Mean Squared Error for training data using closed form solution with lambda= 70:
0.27459068601841896
Mean Squared Error for training data using closed form solution with lambda= 71:
0.2746189420491818
Mean Squared Error for training data using closed form solution with lambda= 72:
0.2746474612077288
Mean Squared Error for training data using closed form solution with lambda= 73:
0.2746762411598135
Mean Squared Error for training data using closed form solution with lambda= 74:
0.2747052796079727
Mean Squared Error for training data using closed form solution with lambda= 75:
0.27473457429068865
Mean Squared Error for training data using closed form solution with lambda= 76:
0.27476412298157504
Mean Squared Error for training data using closed form solution with lambda= 77:
0.27479392348858406
Mean Squared Error for training data using closed form solution with lambda= 78:
0.27482397365323746
Mean Squared Error for training data using closed form solution with lambda= 79:
0.274854271349877
Mean Squared Error for training data using closed form solution with lambda= 80:
0.2748848144849361
Mean Squared Error for training data using closed form solution with lambda= 81:
0.2749156009962318
Mean Squared Error for training data using closed form solution with lambda= 82:
0.274946628852275
Mean Squared Error for training data using closed form solution with lambda= 83:
0.27497789605160017
Mean Squared Error for training data using closed form solution with lambda= 84:
0.2750094006221126
```

```
Mean Squared Error for training data using closed form solution with lambda= 85:
0.27504114062045326
Mean Squared Error for training data using closed form solution with lambda= 86:
0.27507311413138086
Mean Squared Error for training data using closed form solution with lambda= 87:
0.27510531926716997
Mean Squared Error for training data using closed form solution with lambda= 88:
0.27513775416702535
Mean Squared Error for training data using closed form solution with lambda= 89:
0.27517041699651146
Mean Squared Error for training data using closed form solution with lambda= 90:
0.27520330594699766
Mean Squared Error for training data using closed form solution with lambda= 91:
0.27523641923511644
Mean Squared Error for training data using closed form solution with lambda= 92:
0.27526975510223745
Mean Squared Error for training data using closed form solution with lambda= 93:
0.2753033118139544
Mean Squared Error for training data using closed form solution with lambda= 94:
0.2753370876595843
Mean Squared Error for training data using closed form solution with lambda= 95:
0.27537108095168217
Mean Squared Error for training data using closed form solution with lambda= 96:
0.27540529002556463
Mean Squared Error for training data using closed form solution with lambda= 97:
0.27543971323884886
Mean Squared Error for training data using closed form solution with lambda= 98:
0.27547434897100154
Mean Squared Error for training data using closed form solution with lambda= 99:
0.27550919562289927
Mean Squared Error for training data using closed form solution with lambda=
100:  0.2755442516164008


Minimum MSE value for the training data =  0.2734668788542625 obtained for
lambda = 1
```



[27]:
```
MSE_test = []
lambda_list = []
#for i in np.arange(1e-5,10.0,1e-2):
for i in range(1,101,1):
    lambda_list.append(i)
    theta = ClosedForm_RidgeRegression(X_train,Y_train,i)
    Y_pred = predict(theta,X_test)
    print('Mean Squared Error for testing data using closed form solution with␣
 ↪lambda= '+ str(i) + ': ',
          metrics.mean_squared_error(Y_test, Y_pred))
    MSE_test.append(metrics.mean_squared_error(Y_test, Y_pred))
print("\n")
min_lambda = lambda_list[MSE_test.index(min(MSE_test))]
print("Minimum MSE value for the testing data = ", min(MSE_test), "obtained for␣
 ↪lambda =", min_lambda)
```

```
Mean Squared Error for testing data using closed form solution with lambda= 1:
0.5005925887198549
Mean Squared Error for testing data using closed form solution with lambda= 2:
0.5006683305125841
Mean Squared Error for testing data using closed form solution with lambda= 3:
```

```
0.5007445752685044
Mean Squared Error for testing data using closed form solution with lambda= 4:
0.5008213094525349
Mean Squared Error for testing data using closed form solution with lambda= 5:
0.500898520001577
Mean Squared Error for testing data using closed form solution with lambda= 6:
0.5009761943057558
Mean Squared Error for testing data using closed form solution with lambda= 7:
0.5010543201904955
Mean Squared Error for testing data using closed form solution with lambda= 8:
0.501132885899432
Mean Squared Error for testing data using closed form solution with lambda= 9:
0.50121188007802
Mean Squared Error for testing data using closed form solution with lambda= 10:
0.5012912917578864
Mean Squared Error for testing data using closed form solution with lambda= 11:
0.5013711103418312
Mean Squared Error for testing data using closed form solution with lambda= 12:
0.5014513255895406
Mean Squared Error for testing data using closed form solution with lambda= 13:
0.5015319276038572
Mean Squared Error for testing data using closed form solution with lambda= 14:
0.5016129068176468
Mean Squared Error for testing data using closed form solution with lambda= 15:
0.5016942539812503
Mean Squared Error for testing data using closed form solution with lambda= 16:
0.5017759601504561
Mean Squared Error for testing data using closed form solution with lambda= 17:
0.5018580166749654
Mean Squared Error for testing data using closed form solution with lambda= 18:
0.5019404151873621
Mean Squared Error for testing data using closed form solution with lambda= 19:
0.5020231475925323
Mean Squared Error for testing data using closed form solution with lambda= 20:
0.502106206057529
Mean Squared Error for testing data using closed form solution with lambda= 21:
0.5021895830018502
Mean Squared Error for testing data using closed form solution with lambda= 22:
0.5022732710881335
Mean Squared Error for testing data using closed form solution with lambda= 23:
0.5023572632132074
Mean Squared Error for testing data using closed form solution with lambda= 24:
0.5024415524995294
Mean Squared Error for testing data using closed form solution with lambda= 25:
0.5025261322869632
Mean Squared Error for testing data using closed form solution with lambda= 26:
0.5026109961248924
Mean Squared Error for testing data using closed form solution with lambda= 27:
```

```
0.5026961377646414
Mean Squared Error for testing data using closed form solution with lambda= 28:
0.5027815511522193
Mean Squared Error for testing data using closed form solution with lambda= 29:
0.5028672304213353
Mean Squared Error for testing data using closed form solution with lambda= 30:
0.5029531698866975
Mean Squared Error for testing data using closed form solution with lambda= 31:
0.5030393640375841
Mean Squared Error for testing data using closed form solution with lambda= 32:
0.5031258075316567
Mean Squared Error for testing data using closed form solution with lambda= 33:
0.5032124951890279
Mean Squared Error for testing data using closed form solution with lambda= 34:
0.5032994219865451
Mean Squared Error for testing data using closed form solution with lambda= 35:
0.5033865830523175
Mean Squared Error for testing data using closed form solution with lambda= 36:
0.5034739736604354
Mean Squared Error for testing data using closed form solution with lambda= 37:
0.5035615892259011
Mean Squared Error for testing data using closed form solution with lambda= 38:
0.5036494252997626
Mean Squared Error for testing data using closed form solution with lambda= 39:
0.5037374775644202
Mean Squared Error for testing data using closed form solution with lambda= 40:
0.5038257418291154
Mean Squared Error for testing data using closed form solution with lambda= 41:
0.503914214025602
Mean Squared Error for testing data using closed form solution with lambda= 42:
0.5040028902039613
Mean Squared Error for testing data using closed form solution with lambda= 43:
0.5040917665285937
Mean Squared Error for testing data using closed form solution with lambda= 44:
0.5041808392743496
Mean Squared Error for testing data using closed form solution with lambda= 45:
0.5042701048228084
Mean Squared Error for testing data using closed form solution with lambda= 46:
0.5043595596586953
Mean Squared Error for testing data using closed form solution with lambda= 47:
0.5044492003664289
Mean Squared Error for testing data using closed form solution with lambda= 48:
0.5045390236268023
Mean Squared Error for testing data using closed form solution with lambda= 49:
0.5046290262137791
Mean Squared Error for testing data using closed form solution with lambda= 50:
0.5047192049914125
Mean Squared Error for testing data using closed form solution with lambda= 51:
```

```
0.5048095569108725
Mean Squared Error for testing data using closed form solution with lambda= 52:
0.5049000790075835
Mean Squared Error for testing data using closed form solution with lambda= 53:
0.5049907683984668
Mean Squared Error for testing data using closed form solution with lambda= 54:
0.5050816222792798
Mean Squared Error for testing data using closed form solution with lambda= 55:
0.5051726379220536
Mean Squared Error for testing data using closed form solution with lambda= 56:
0.5052638126726183
Mean Squared Error for testing data using closed form solution with lambda= 57:
0.5053551439482235
Mean Squared Error for testing data using closed form solution with lambda= 58:
0.5054466292352361
Mean Squared Error for testing data using closed form solution with lambda= 59:
0.5055382660869254
Mean Squared Error for testing data using closed form solution with lambda= 60:
0.5056300521213241
Mean Squared Error for testing data using closed form solution with lambda= 61:
0.5057219850191623
Mean Squared Error for testing data using closed form solution with lambda= 62:
0.5058140625218792
Mean Squared Error for testing data using closed form solution with lambda= 63:
0.5059062824297016
Mean Squared Error for testing data using closed form solution with lambda= 64:
0.5059986425997876
Mean Squared Error for testing data using closed form solution with lambda= 65:
0.5060911409444377
Mean Squared Error for testing data using closed form solution with lambda= 66:
0.5061837754293654
Mean Squared Error for testing data using closed form solution with lambda= 67:
0.5062765440720337
Mean Squared Error for testing data using closed form solution with lambda= 68:
0.506369444940034
Mean Squared Error for testing data using closed form solution with lambda= 69:
0.5064624761495409
Mean Squared Error for testing data using closed form solution with lambda= 70:
0.5065556358638036
Mean Squared Error for testing data using closed form solution with lambda= 71:
0.5066489222916962
Mean Squared Error for testing data using closed form solution with lambda= 72:
0.5067423336863143
Mean Squared Error for testing data using closed form solution with lambda= 73:
0.5068358683436214
Mean Squared Error for testing data using closed form solution with lambda= 74:
0.5069295246011394
Mean Squared Error for testing data using closed form solution with lambda= 75:
```

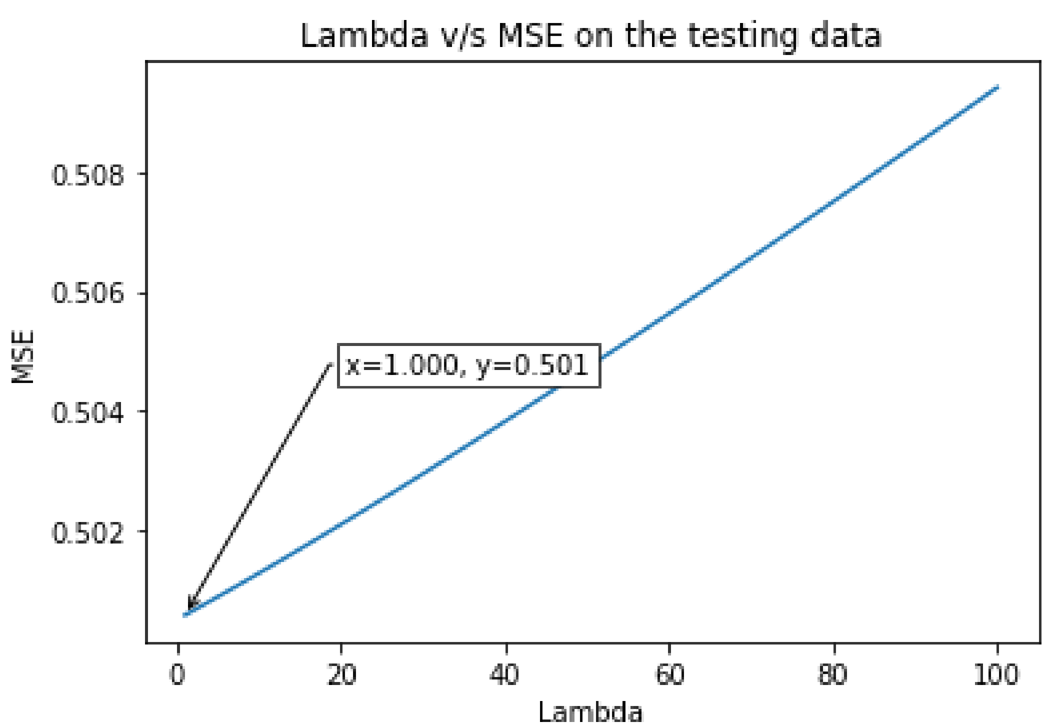
```
0.5070233008366813
Mean Squared Error for testing data using closed form solution with lambda= 76:
0.5071171954671343
Mean Squared Error for testing data using closed form solution with lambda= 77:
0.5072112069472702
Mean Squared Error for testing data using closed form solution with lambda= 78:
0.5073053337686099
Mean Squared Error for testing data using closed form solution with lambda= 79:
0.5073995744583145
Mean Squared Error for testing data using closed form solution with lambda= 80:
0.5074939275781184
Mean Squared Error for testing data using closed form solution with lambda= 81:
0.5075883917232973
Mean Squared Error for testing data using closed form solution with lambda= 82:
0.5076829655216698
Mean Squared Error for testing data using closed form solution with lambda= 83:
0.5077776476326278
Mean Squared Error for testing data using closed form solution with lambda= 84:
0.5078724367462084
Mean Squared Error for testing data using closed form solution with lambda= 85:
0.5079673315821837
Mean Squared Error for testing data using closed form solution with lambda= 86:
0.5080623308891917
Mean Squared Error for testing data using closed form solution with lambda= 87:
0.5081574334438861
Mean Squared Error for testing data using closed form solution with lambda= 88:
0.5082526380501177
Mean Squared Error for testing data using closed form solution with lambda= 89:
0.5083479435381452
Mean Squared Error for testing data using closed form solution with lambda= 90:
0.5084433487638663
Mean Squared Error for testing data using closed form solution with lambda= 91:
0.508538852608074
Mean Squared Error for testing data using closed form solution with lambda= 92:
0.5086344539757397
Mean Squared Error for testing data using closed form solution with lambda= 93:
0.5087301517953179
Mean Squared Error for testing data using closed form solution with lambda= 94:
0.5088259450180703
Mean Squared Error for testing data using closed form solution with lambda= 95:
0.5089218326174174
Mean Squared Error for testing data using closed form solution with lambda= 96:
0.5090178135883017
Mean Squared Error for testing data using closed form solution with lambda= 97:
0.5091138869465809
Mean Squared Error for testing data using closed form solution with lambda= 98:
0.5092100517284336
Mean Squared Error for testing data using closed form solution with lambda= 99:
```

```
0.5093063069897821
Mean Squared Error for testing data using closed form solution with lambda= 100:
0.5094026518057427


Minimum MSE value for the testing data =  0.5005925887198549 obtained for lambda
= 1
```

[28]:
```
def annot_min(x,y, ax=None):
    xmin = x[np.argmin(y)]
    ymin = min(y)
    text= "x={:.3f}, y={:.3f}".format(xmin, ymin)
    if not ax:
        ax=plt.gca()
    bbox_props = dict(boxstyle="square,pad=0.3", fc="w", ec="k", lw=0.72)
    arrowprops=dict(arrowstyle="->",connectionstyle="angle,angleA=0,angleB=60")
    kw = dict(xycoords='data',textcoords="axes fraction",
              arrowprops=arrowprops, bbox=bbox_props, ha="right", va="top")
    ax.annotate(text, xy=(xmin, ymin), xytext=(0.50,0.50), **kw)

plt.plot(lambda_list, MSE_test)
annot_min(lambda_list, MSE_test)
plt.title("Lambda v/s MSE on the testing data")
plt.xlabel("Lambda")
plt.ylabel("MSE")
plt.show()
```



The MSE value for the training data set obtained in the Ridge regression is almost similar to the value obtained in the Linear Regression. The MSE value for the testing data set obtained in the Ridge regression is slightly less compared to those obtained in the Linear Regression.

Problem 5

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} \\ 1 & x_{21} & x_{22} \\ \vdots & \vdots & \vdots \\ 1 & x_{N1} & x_{N2} \end{bmatrix}_{N\times 3} = \begin{bmatrix} 1 & x_{11} & 2x_{11} \\ 1 & x_{21} & 2x_{21} \\ \vdots & \vdots & \vdots \\ 1 & x_{N1} & 2x_{N1} \end{bmatrix}_{N\times 3} \quad \left[\because x_{i2} = 2\cdot x_{i1}\right]$$

$$X^T = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_{11} & x_{21} & x_{31} & \cdots & x_{N1} \\ 2x_{11} & 2x_{21} & 2x_{31} & \cdots & 2x_{N1} \end{bmatrix}_{3\times N}$$

$$\therefore X^TX = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_{11} & x_{21} & x_{31} & \cdots & x_{N1} \\ 2x_{11} & 2x_{21} & 2x_{31} & \cdots & 2x_{N1} \end{bmatrix} \begin{bmatrix} 1 & x_{11} & 2x_{11} \\ 1 & x_{21} & 2x_{21} \\ \vdots & \vdots & \vdots \\ 1 & x_{N1} & 2x_{N1} \end{bmatrix}$$

$$= \begin{bmatrix} N & x_{11}+x_{21}+\cdots+x_{N1} & 2(x_{11}+x_{21}+\cdots+x_{N1}) \\ x_{11}+x_{21}+\cdots x_{N1} & x_{11}^2+x_{21}^2+\cdots+x_{N1}^2 & 2(x_{11}^2+x_{21}^2+\cdots+x_{N1}^2) \\ 2(x_{11}+x_{21}+ +x_{N1}) & 2(x_{11}^2+x_{21}^2+\cdots+x_{N1}^2) & 4(x_{11}^2+x_{21}^2+\cdots+x_{N1}^2) \end{bmatrix}_{3\times 3}$$

$$\begin{array}{cccc} & C1 & C2 & C3 \\ = & N & x_{11}+x_{21}+\cdots+x_{N1} & 2(x_{11}+x_{21}+\cdots+x_{N1}) \\ & x_{11}+x_{21}+\cdots x_{N1} & x_{11}^2+x_{21}^2+\cdots+x_{N1}^2 & 2(x_{11}^2+x_{21}^2+\cdots+x_{N1}^2) \\ & 2(x_{11}+x_{21}+\cdots+x_{N1}) & 2(x_{11}^2+x_{21}^2+\cdots+x_{N1}^2) & 4(x_{11}^2+x_{21}^2+\cdots+x_{N1}^2) \end{array}_{3\times 3}$$

Since column 2 and column 3 are dependent as $C3 = 2C2$, hence ~~they~~ matrix X^TX is not invertible.

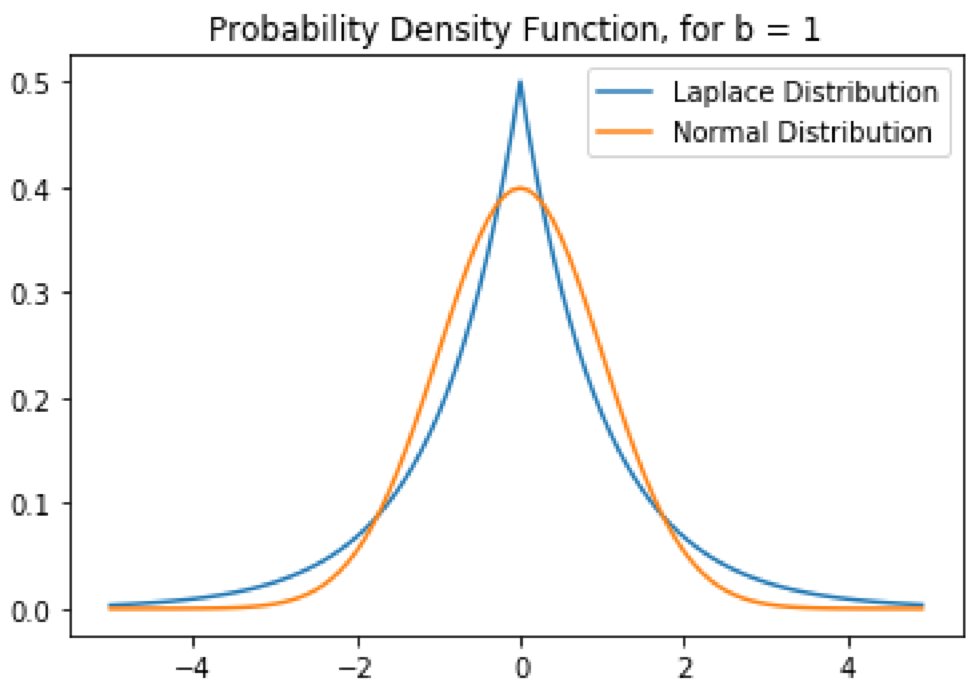
The Rank of the matrix $(X^TX) = 2$.

Scanned by CamScanner

5 Problem 6

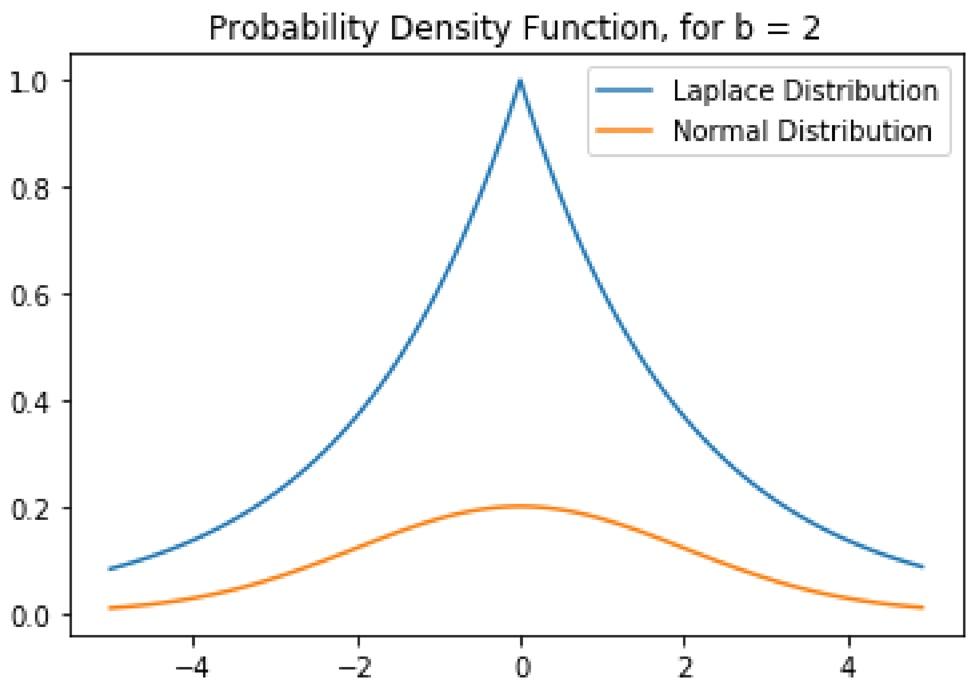
5.1 6a

[29]:
```
Laplace = []
Normal = []
x_val = np.arange(-5,5,0.1)
b = 1
for x in x_val:
        Laplace.append((1/2*b) * math.exp(-abs(x)/b))
        Normal.append((1/(b * np.sqrt(2*np.pi)) * np.exp(-(x)**2/(2 * b**2))))
plt.plot(x_val,Laplace, label = "Laplace Distribution")
plt.plot(x_val,Normal, label = "Normal Distribution")
plt.legend()
plt.title("Probability Density Function, for b = 1")
plt.show()
```

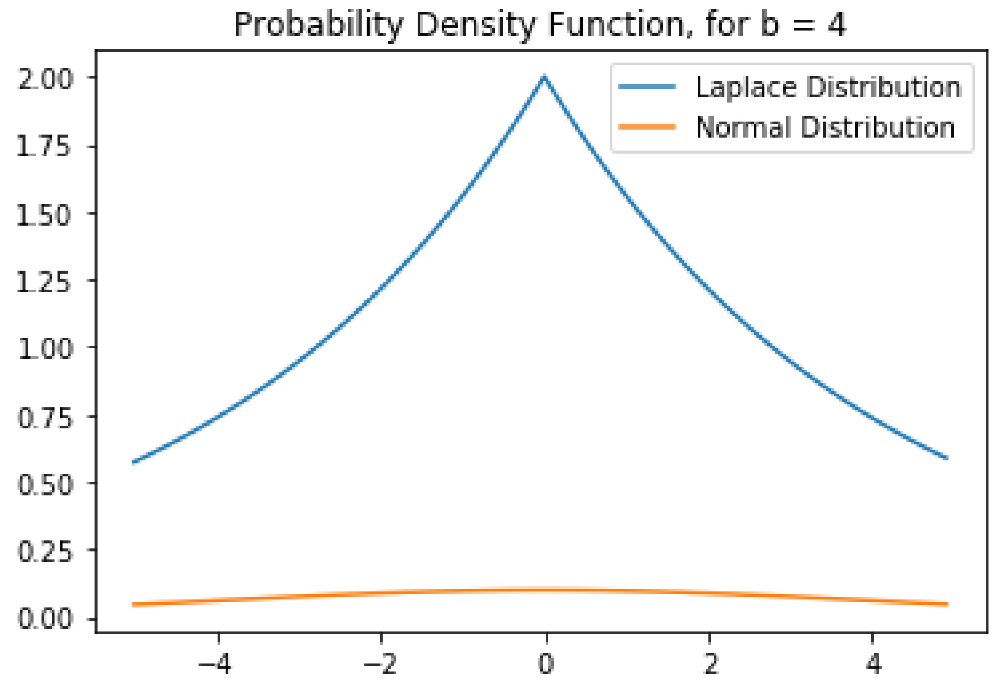


[30]:
```
Laplace = []
Normal = []
b = 2
for x in x_val:
        Laplace.append((1/2*b) * math.exp(-abs(x)/b))
        Normal.append((1/(b * np.sqrt(2*np.pi)) * np.exp(-(x)**2/(2 * b**2))))
plt.plot(x_val,Laplace, label = "Laplace Distribution")
plt.plot(x_val,Normal, label = "Normal Distribution")
```

```
plt.legend()
plt.title("Probability Density Function, for b = 2")
plt.show()
```



[31]:
```
Laplace = []
Normal = []
b = 4
for x in x_val:
        Laplace.append((1/2*b) * math.exp(-abs(x)/b))
        Normal.append((1/(b * np.sqrt(2*np.pi)) * np.exp(-(x)**2/(2 * b**2))))
plt.plot(x_val,Laplace, label = "Laplace Distribution")
plt.plot(x_val,Normal, label = "Normal Distribution")
plt.legend()
plt.title("Probability Density Function, for b = 4")
plt.show()
```



Normal distribution have very short tails whereas the Laplace distribution have longer tails because the Laplace density is expressed in terms of the absolute difference from the mean.

Problem 6

b) $y_i = h_\theta(x_i) + \varepsilon$.

$\varepsilon \sim \text{Laplace}(0, b)$ with pdf $p(x) = \frac{1}{2b} e^{-|x|/b}$

$y_i | x_i \sim \text{Laplace}(0, b)$

$$f(y_i | x_i ; \theta, b) = \cancel{\frac{1}{2b} e^{-|x|/b}} = \frac{1}{2b} e^{-|y_i - h_\theta(x_i)| / b}$$

↳ PDF of the Noise.

Training dataset,

$$f(y_1, \ldots y_N | x_1, \ldots, x_N ; \theta, b) = \prod_{i=1}^{N} f(y_i | x_i ; \theta, b)$$

The likelihood function,

$$\text{Max } L(\theta) = \prod_{i=1}^{N} P[y_i | x_i ; \theta] = \prod_{i=1}^{N} f(y_i | x_i ; \theta, b)$$

[~~Assuming~~, training points are independent]

$$L(\theta) = \prod_{i=1}^{N} \frac{1}{2b} e^{-|y_i - h_\theta(x_i)| / b}$$

$$= \left(\frac{1}{2b}\right)^N \prod_{i=1}^{N} e^{-|y_i - h_\theta(x_i)| / b}$$

$$\text{Log} L(\theta) = \underbrace{-N \log(2b)}_{\text{constant}} - \sum_{i=1}^{N} |y_i - h_\theta(x_i)| / b \rightarrow \text{const}$$

Maximizing the log likelihood is to minimize the term.

$$\sum_{i=1}^{N} |y_i - h_\theta(x_i)|$$

$$\therefore J(\theta) = \sum_{i=1}^{N} |y_i - h_\theta(x_i)|$$

Problem 6

c) $J(\theta)$ for ~~N~~ Normal Noise.

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} [h_\theta(x_i) - y_i]^2 \Rightarrow \text{Error is squared,}$$

$J(\theta)$ for Laplace Noise

$$J(\theta) = \sum_{i=1}^{N} |h_\theta(x_i) - y_i| \Rightarrow \text{Absolute value of the error}$$

Considering an outlier in the training data, defined as a point of a high residual, the objective with Laplace Noise is more resilient to the effect of outliers since the $J(\theta)$ for Normal Noise is more sensitive to residuals ~~with~~ ~~square~~ as the residuals are squared; whereas the residuals are not squared in Laplace making it more resilient. ~~the~~