Generate 100 real number for the variable X from the uniform distribution U [0,1].

In [698…
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import *
import math
from sklearn.metrics import mean_squared_error
import random
import math
```

In [699…
```
x_train_data = np.random.uniform(0.0,1.0,100)
x_train_data = np.sort(x_train_data)
print(x_train_data)
```

```
[0.01794214 0.02744726 0.05441816 0.05867652 0.0713818  0.07333642
 0.07607485 0.08807077 0.0905154  0.09557764 0.10174471 0.10392102
 0.12581895 0.12675826 0.1302724  0.13215192 0.14958819 0.15863186
 0.1615118  0.17024829 0.17375638 0.18158474 0.18173979 0.18731
 0.19138408 0.20731153 0.22371733 0.23958667 0.24523815 0.24859255
 0.2591817  0.26414141 0.26585066 0.31148471 0.3156191  0.36105344
 0.36472357 0.3664542  0.3690592  0.3763201  0.37675359 0.37845234
 0.38186884 0.38656592 0.39855708 0.40691913 0.40978811 0.41858403
 0.43320326 0.43526472 0.43534413 0.43561907 0.46098437 0.46450069
 0.48885864 0.50290865 0.50691583 0.51887929 0.54086965 0.55879466
 0.57721515 0.58252319 0.59030779 0.63047703 0.66859332 0.67077571
 0.69667406 0.69878697 0.69997343 0.71051316 0.7711821  0.77303307
 0.77316349 0.78376263 0.78606261 0.79150923 0.80335978 0.80737344
 0.81041737 0.81408864 0.81915814 0.82113865 0.83063635 0.84552814
 0.85335399 0.85377867 0.87257566 0.89354976 0.91044476 0.94307996
 0.94558516 0.94851546 0.95434324 0.96386722 0.97288813 0.97848836
 0.99236544 0.99294035 0.99390155 0.99436573]
```

Construct the training set T = { (x1,y1),(x2,y2),.....,(x100,y100)} using the relation

Yi = sin(2 π xi) + εi where εi ~ N(0,0.25).

In [700…
```
#finding mean
import statistics
x_mean = statistics.mean(x_train_data) # calculating mean of x_train
print(x_mean)
V = np.zeros(len(x_train_data))
for i in range(len(x_train_data)):
```

```
  V[i] = x_train_data[i] - x_mean
V = np.square(V)

#finding standard deviation
Standard_deviation = np.sum(V) / len(V)  # calculating standard deviation
print(Standard_deviation)

#function for calculating epilson (normal distribution)
def epilson_value(std,mean,x):
  return (np.pi*std) * np.exp(-0.5*((x-mean)/std)**2)
```

```
0.49108126601378493
0.092181603966571
```

In [701…

```
y_train_data = np.zeros(len(x_train_data))
for i in range (len(x_train_data)):
  y_train_data[i] = np.sin(2 * np.pi * x_train_data[i]) + epilson_value(0.25,0,x_train_data[i])
print(y_train_data)
```

```
[ 0.89587326  0.95228161  1.1023062   1.12444096  1.18764576  1.19697497
  1.20986143  1.26369714  1.27412878  1.29512937  1.31959367  1.32792723
  1.40270933  1.40553651  1.41582782  1.42114573  1.46416091  1.48187685
  1.48684641  1.49991293  1.50430332  1.51231692  1.51245067  1.51660951
  1.51885364  1.52113343  1.51265546  1.49405907  1.48499298  1.47901079
  1.45721677  1.44550548  1.44124963  1.28771051  1.27019237  1.04308256
  1.02222758  1.01227724  0.99716192  0.95418376  0.95157952  0.94133359
  0.92053453  0.89153211  0.8154879   0.76090065  0.74190334  0.68287855
  0.5824979   0.56814683  0.56759315  0.56567565  0.38616622  0.36098858
  0.18603178  0.08556567  0.05709277 -0.02721247 -0.17834733 -0.29647482
 -0.41170703 -0.44357062 -0.48910777 -0.69835614 -0.85003627 -0.857174
 -0.92821848 -0.93287755 -0.93541712 -0.95554003 -0.98441405 -0.9829562
 -0.98284831 -0.97181758 -0.96883706 -0.96095198 -0.93982613 -0.93145582
 -0.92470383 -0.91610086 -0.90340448 -0.89818961 -0.87122621 -0.82263322
 -0.79413394 -0.79253095 -0.71601812 -0.61876003 -0.53242969 -0.34942549
 -0.3346618  -0.31728667 -0.28241346 -0.22461891 -0.16912206 -0.1343801
 -0.04765343 -0.04404766 -0.0380179  -0.0351055 ]
```

In the similar way construct a testing set of size 50 I,e. Test = { (x'1,y'1),(x'2,y'2),.....,(x'50,y'50)}.

In [702…

```
#constructing x test data  in similar way
x_test_data = np.random.uniform(0.0,1.0,50)
x_test_data = np.sort(x_test_data)
```

```
#constructing y test data
y_test_data = np.zeros(len(x_test_data))
for i in range (len(x_test_data)):
  y_test_data[i] = np.sin(2 * np.pi * x_test_data[i]) + epilson_value(0.25,0,x_test_data[i])
print(y_test_data)
```

```
[ 1.16363364  1.17432418  1.17934609  1.22615422  1.31291815  1.3461175
  1.37313652  1.41373663  1.41400553  1.44672771  1.50504776  1.51621272
  1.49671024  1.44382457  1.41236224  1.29256195  1.23564702  1.2198114
  1.20680701  1.08524103  1.06231092  1.03893084  0.91547461  0.89956443
  0.490673    0.40891108  0.18524542  0.10873482  0.06941494  0.00466303
 -0.03678511 -0.07604463 -0.15246535 -0.18003984 -0.22501651 -0.23925066
 -0.25115603 -0.66492243 -0.76511902 -0.77671098 -0.80887082 -0.98853465
 -0.97610935 -0.97005074 -0.94581327 -0.73384489 -0.63078696 -0.52709649
 -0.4520162  -0.2240428 ]
```

Estimate the regularized Least Squares Polynomial Regression model of order M= 9, using the training set T using direct method. You also need to tune the regularization parameter λ which corresponds to minimum RMSE. After tuning the parameter λ, evaluate your estimated function using NMSE, RMSE, MAE and R2 on test set.

In [703…

```
def get_coef(x,y,m):
  matx=np.zeros((m+1,m+1))
  maty=np.zeros(m+1)
  for i in range(m+1):
    for j in range(m+1):
      if(i==0 and j==0):
        matx[i][j]=len(x)
      else:
        matx[i][j]=np.sum(x**(i+j))
  for i in range(m+1):
    maty[i]=np.sum(y*(x**i))
  maty=np.matrix.transpose(maty)
  X = np.linalg.inv(matx).dot(maty)
  return X
```

In [704…

```
coeff = get_coef(x_train_data, y_train_data, 9)
print("Coefficients for M=9\n", coeff)
```

```
Coefficients for M=9
 [   0.78584679    6.25163644   -5.6101222   -47.87365913   58.46494675
  -11.22441864   62.08065796 -120.89346313   71.25682068  -13.23783875]
```

In [705…

```
# def squared_error(ys_orig,ys_line):
#     return sum((ys_line - ys_orig) * (ys_line - ys_orig))
```

```
# def coefficient_of_determination(ys_orig,ys_line):
#     y_mean_line = [statistics.mean(ys_orig) for y in ys_orig]
#     squared_error_regr = squared_error(ys_orig, ys_line)
#     squared_error_y_mean = squared_error(ys_orig, y_mean_line)
#     return 1 - (squared_error_regr/squared_error_y_mean)
```

In [706…

```
def calculate_rms2(x, y, m): #m=coefficient
  y_predicted = x.dot(m)
  rms = ((np.sum((y_predicted-y)**2))/50)**0.5
  return [rms, y_predicted]

def MAE(y, y_pred, n):
    sum=0
    for i in range(n):
        sum+=abs(y[i] - y_pred[i])
    error=sum/n
    return error

def NMSE(y,y_pred,rmse):
    #(y-y_pred)**2/(yTestmax - yTestmean) --- for normalized mean square errors
    nmse = rmse/(max(y)-min(y))
    return nmse

def rms(x,y,m = [9]):
  y_predicted=np.zeros(len(x))
  c=0
  for i in range(len(x)):
    y_predicted[i]+=m[0]
    for j in range(len(m)-1):
      y_predicted[i]+=m[j+1]*x[i]**j
    c+=(y_predicted[i]-y[i])**2
    c=c**0.5/len(x)
  return [c,y_predicted]
```

In [707…

```
train_9 = calculate_rms(x_train_data, y_train_data, coeff)
train_9
```

Out[707]:
```
[0.000123006989355617,
 array([ 6.92175074,  6.84863845,  6.59977049,  6.55519137,  6.41416194,
         6.39143011,  6.35913124,  6.2116038 ,  6.18036825,  6.11446877,
         6.0320221 ,  6.00237393,  5.68891208,  5.67488135,  5.62198658,
         5.59343918,  5.32051388,  5.17360168,  5.12610912,  4.9800669 ,
         4.92062852,  4.78645147,  4.7837733 ,  4.6870533 ,  4.61571064,
         4.33240668,  4.03441179,  3.74163471,  3.63656235,  3.57404028,
         3.37606925,  3.28311306,  3.25105493,  2.39711118,  2.32051143,
         1.49841228,  1.43405928,  1.40384361,  1.3585227 ,  1.23325829,
         1.22583059,  1.19677999,  1.13863172,  1.0593113 ,  0.8602615 ,
         0.72456625,  0.67862987,  0.53985195,  0.31640355,  0.28565152,
         0.28447077,  0.28038486, -0.08125097, -0.1288951 , -0.44122122,
        -0.60670726, -0.6518656 , -0.78118183, -0.99686732, -1.15104403,
        -1.28883006, -1.32461404, -1.3739093 , -1.56814525, -1.66081376,
        -1.66349626, -1.67445057, -1.67368184, -1.67314276, -1.66500378,
        -1.50970008, -1.502318  , -1.50179248, -1.45678076, -1.44642638,
        -1.42110045, -1.36223797, -1.34118717, -1.32486252, -1.30477106,
        -1.27632569, -1.26499785, -1.20906991, -1.11643521, -1.06557803,
        -1.06277878, -0.93523884, -0.78597943, -0.66199668, -0.41790905,
        -0.39911294, -0.37714122, -0.33351126, -0.26250599, -0.19574492,
        -0.15461351, -0.05400099, -0.04987857, -0.04299523, -0.03967519])]
```

In [708…
```
def regularization(x, y, l):
  identity = np.identity(x.shape[1])
  F = np.linalg.inv((x.T).dot(x)+l*identity).dot((x.T).dot(y))
  return F
```

In [709…
```
def plot_reg_graph(x, y1, y2, y3, y4, m, l):
  plt.title(f"Graph for m={m} and lambda={l}")
  plt.scatter(x, y1)
  plt.plot(x, y1, label="Actual")
  plt.plot(x, y2, label="Predicted")
  plt.plot(x, y3, label="Noise")
  plt.plot(x, y4, label="Regularized")
  plt.legend()
  plt.show()
```

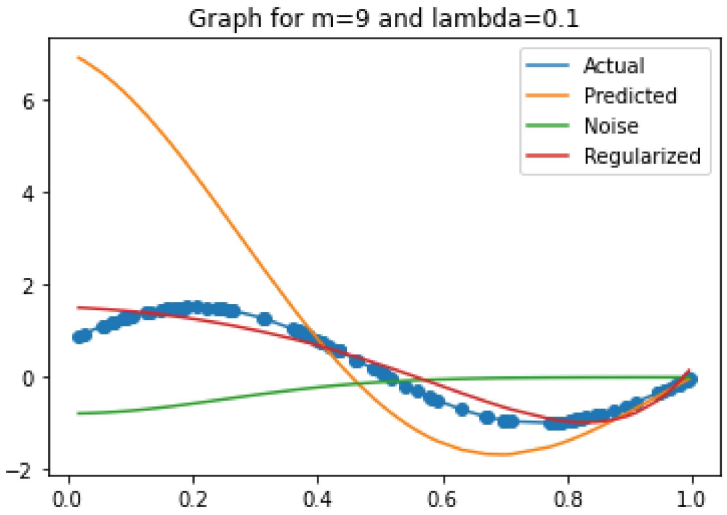
In [710…
```
#function to calculate error/noise
def calculate_error(x, y):
  err = np.zeros(len(y))
  for i in range(len(y)):
    err[i] = err[i] - epilson_value(0.25, 0, x[i])
  return err
```

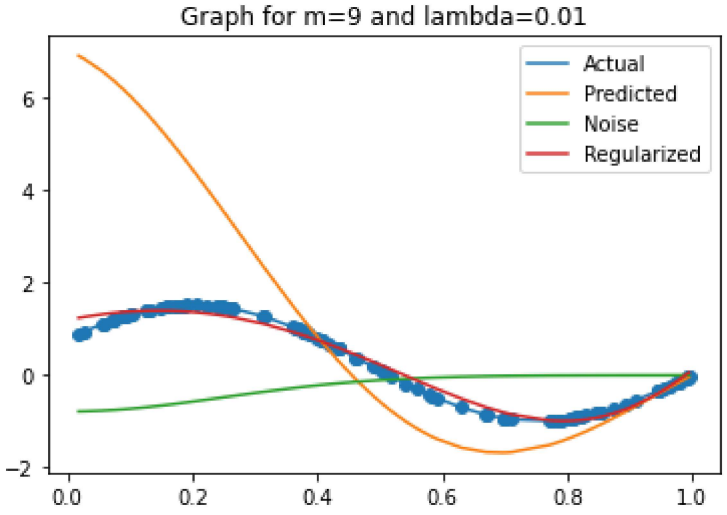
```
y_train_error = calculate_error(x_train_data, y_train_data)
y_test_error = calculate_error(x_test_data, y_test_data)
```

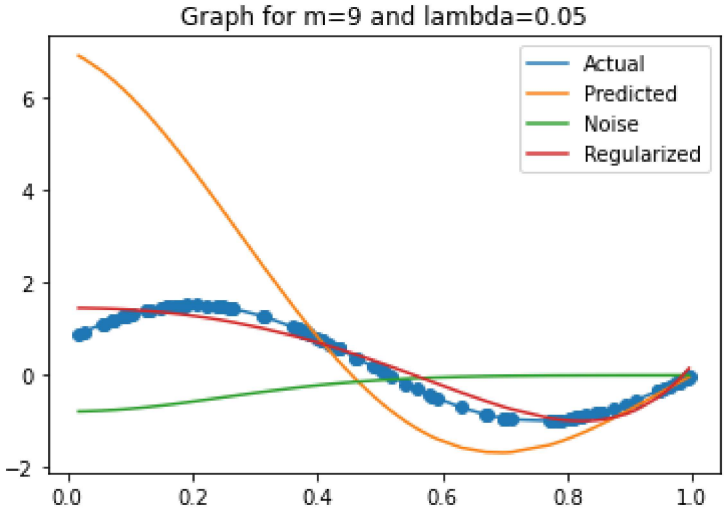
In [711…
```
x1 = []
x1.append(np.asarray([x_train_data//x_train_data, x_train_data, x_train_data**2, x_train_data**3, x_train_data**4, x_tr

y1 = np.asarray([y_train_data]).T
```

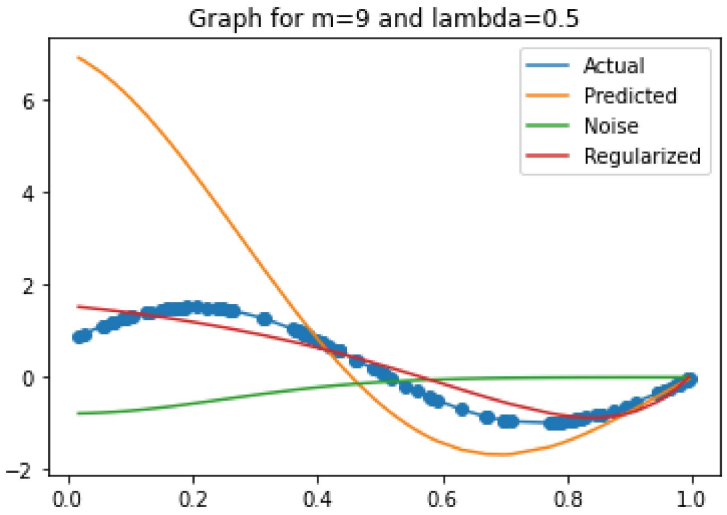
In [712…
```
#Todo: Calculate it for different lambda values

l = 0.001
#finding the coefficient
coeff_r = []

coeff_r.append(regularization(x1[0], y1, l))

train_reg = []
train_reg.append(calculate_rms2(x1[0], y1, coeff_r[0]))
```

In [713…
```
#plotting graph for different lambda values
lambda_values = [0.1, 0.01, 0.05, 0.5]
for l in lambda_values:
  coef_r = regularization(x1[0], y1, l)
  rmse = calculate_rms2(x1[0],y1, coef_r)
  plot_reg_graph(x_train_data, y_train_data, train_9[1], y_train_error, rmse[1], 9, l)
```




Graph for m=9 and lambda=0.01
Actual
Predicted
Noise
Regularized
6
4
2
0
−2
0.0
0.2
0.4
0.6
0.8
1.0


Graph for m=9 and lambda=0.05
Actual
Predicted
Noise
Regularized
6
4
2
0
−2
0.0
0.2
0.4
0.6
0.8
1.0



In [714…
```
def r2score(y,y_pred):
  return((y_test_data - y_predicted)**2).sum() / ((y_test_data - y_test_data.mean())**2).sum()
```

In [715…
```
rmse, y_predicted = rms(x_test_data, y_test_data,[9])
print("RMSE:", rmse)

mae = MAE(y_test_data,y_predicted,50)
print("MAE:",mae)

nmse = NMSE(y_test_data,y_predicted,rmse)
print("NMSE:",nmse)

# r2 = ((y_test_data - y_predicted)**2).sum() / ((y_test_data - y_test_data.mean())**2).sum()
r2 = r2score(y_test_data,y_predicted)
print("R2 score: ",r2)
```

```
RMSE: 0.18468590374900884
MAE: 8.57952855002833
NMSE: 0.07373434386787023
R2 score:  96.42058020725977
```

Estimate the regularized Least Squares Polynomial Regression model of order M= 9, on the training set T using gradient descent method with the tuned value of λ obtained in question 3 by selecting an appropriate step length η. Compare the solution obtained by direct method and gradient descent method. Evaluate your estimated function using NMSE, RMSE, MAE and R2 on test set.

In [716…
```
#Gradient descent
```

```
def gradient_descent(x,y):
    m_curr = 0
    b_curr = 0
    iterations = 10
    n = len(x)
    learning_rate = 0.08


    for i in range(iterations):
      y_p = m_curr * x + b_curr
      cost = 1/n * np.sum([val**2 for val in (y - y_p)]) #MSE
      md = -(2/n) * np.sum(x * (y - y_p))
      bd = -(2/n) * np.sum(y-y_p)
      m_curr = m_curr - learning_rate * md
      b_curr = b_curr - learning_rate * bd
      print(f"m: {m_curr} b: {b_curr} i: {i} cost: {cost}")

gradient_descent(x_train_data,y_train_data)
```

```
m: -0.01097221838516225 b: 0.058059149704075795 i: 0 cost: 0.996436882524626
m: -0.025921117566988755 b: 0.1076909555987899 i: 1 cost: 0.9555815233020161
m: -0.04397244041864877 b: 0.15055625244406556 i: 2 cost: 0.9238058030446967
m: -0.06442905490410823 b: 0.1879814484307718 i: 3 cost: 0.8979508341801953
m: -0.08673523024961295 b: 0.2210259506819851 i: 4 cost: 0.8760000580953643
m: -0.1104481162620117 b: 0.2505359957455796 i: 5 cost: 0.856665101023785
m: -0.13521497341497124 b: 0.2771876262524085 i: 6 cost: 0.8391217997705306
m: -0.16075499416874842 b: 0.30152100220868205 i: 7 cost: 0.8228419446919657
m: -0.1868447897872496 b: 0.32396779412807714 i: 8 cost: 0.8074860322302287
m: -0.21330680357323914 b: 0.3448730529183489 i: 9 cost: 0.7928349045533252
```

Estimate the regularized Least Squares Polynomial Regression model of order M= 9 on the training set T using stochastic gradient descent method with the tuned value of λ obtained in question 3 by selecting an appropriate step lengths ηi. Compare the solution with the solution obtained by the direct method and gradient descent method. Evaluate your estimated function using NMSE, RMSE, MAE and R2 on test set. Also obtain the norm of the obtained gradient at each iteration (epoch) of stochastic gradient descent method and plot it.

In [717…

```
def sgd_regressor(X, y, learning_rate=0.2, n_epochs=1000, k=40):

    w = np.random.randn(1,9)  # Randomly initializing weights
    b = np.random.randn(1,1)   # Random intercept value

    epoch=1
```

```
    while epoch <= n_epochs:
        temp = np.array(random.sample(X, k))
        # temp = X.sample(k)

        X_tr = temp[:,0:9]
        # print(X_tr)
        y_tr = temp[:,-1]
        # print(y_tr)

        Lw = w
        Lb = b

        loss = 0
        y_pred = []
        sq_loss = []

        for i in range(k):

            Lw = (-2/k * X_tr[i]) * (y_tr[i] - np.dot(X_tr[i],w.T) - b)
            Lb = (-2/k) * (y_tr[i] - np.dot(X_tr[i],w.T) - b)

            w = w - learning_rate * Lw
            b = b - learning_rate * Lb

            y_predicted = np.dot(X_tr[i],w.T)
            y_pred.append(y_predicted.tolist())

        loss = mean_squared_error(y_pred, y_tr)

        # print("Epoch: %d, Loss: %.3f" %(epoch, loss))
        epoch+=1
        learning_rate = learning_rate/1.02
    # print(y_pred)



    return w,b,y_pred
```

In [718…

```
def predict(x,w,b):
    y_pred=[]
    for i in range(len(x)):

        temp_ = x
        X_test = temp_[:]
        y = np.asscalar(np.dot(w,X_test[i])+b)
```

```
        y_pred.append(y)
    return np.array(y_pred)
```

In [718…

In [719…
```
x1=[]
x1.append(np.asarray([x_train_data//x_train_data,x_train_data]).T)
y1=np.asarray([y_train_data]).T

x1.append(np.asarray([x_train_data//x_train_data,x_train_data,x_train_data*x_train_data]).T)

x1.append(np.asarray([x_train_data//x_train_data,x_train_data,x_train_data*x_train_data,x_train_data*x_train_data*x_tra

x1.append(np.asarray([x_train_data//x_train_data,x_train_data,x_train_data*x_train_data,x_train_data*x_train_data*x_tra

x1_list = list(x1[3])
```

In [720…
```
x1_l = []
x1_l.append(np.asarray([x_test_data//x_test_data,x_test_data,x_test_data*x_test_data,x_test_data*x_test_data*x_test_dat
x1_list_l = list(x1_l[0])
```

In [721…
```
y_pred_sgd = []
w,b,y_pred_sgd = sgd_regressor(x1_list,y1)
```

In [722…
```
print('Weights: ',w)
print('Bias: ',b)
```

```
Weights:  [[-0.33053009  1.33277309 -2.2614627   0.89436968 -0.90433348 -0.2577546
   1.4408613   1.97182324 -1.02698748]]
Bias:  [[0.17526353]]
```

Estimate the regularized Least Squares Kernel Regression model by using the Gaussian (RBF) kernel which is given by K (xi , xj) = e −||xi−xj || 2 σ2 .

Also, tune the value of the regularization parameter λ and kernel parameter σ and obtain the best possible estimate using NMSE, RMSE, MAE and R2 on test set.

In [723…
```
def rbf_kernel(x,y):
    sigma=1.2
    K = np.exp(-(1/(sigma**2)) *  np.sum((x-y)**2))
    return K
```

```
def kernel_matrix(x1, x2):
    K = np.zeros((x1.shape[0], x2.shape[0]))
    for i in range(x1.shape[0]):
        for j in range(x2.shape[0]):
            K[i,j] = rbf_kernel(x1[i], x2[j])
    return K

def fit_direct(x,y,l):
    K = kernel_matrix(x,x)
    I = np.identity(x.shape[0]+1)

    K = np.insert(K,-1,1, axis=1)

    theta = np.linalg.pinv((K.T @ K) + (l * I)) @ (K.T @ y)

    return K, theta
```

In [724…
```
def predict_direct(xTest, xTrain,theta):
    K = kernel_matrix(xTest, xTrain)
    K = np.insert(K,-1,1,axis=1)

    Y_pred = K @ theta
    return Y_pred
```

In [725…
```
def kernel_rmse(y,y_pred):
    x = np.sqrt(np.mean((y_pred-y)**2))
    return x
```

In [726…
```
u = fit_direct(x_train_data,y_train_data,0.0001)
```

In [727…
```
yTrain_pred = predict_direct(x_train_data,x_train_data,u[1])
```

In [728…
```
yTrain_pred
```

Out[728]:
```
array([ 1.22951731,  1.26165965,  1.33824287,  1.34836217,  1.37537263,
        1.37910621,  1.38414837,  1.40365071,  1.40711068,  1.41372441,
        1.42078105,  1.42300996,  1.43792461,  1.43826142,  1.43930276,
        1.4397184 ,  1.43892746,  1.43525862,  1.4336302 ,  1.42734662,
        1.42425994,  1.41622243,  1.41604733,  1.40935052,  1.40395549,
        1.37891092,  1.34672297,  1.30967657,  1.29513139,  1.28617114,
        1.25632201,  1.24154276,  1.23633388,  1.07685982,  1.06061459,
        0.86557674,  0.84864955,  0.84061444,  0.82845687,  0.7941854 ,
        0.79212198,  0.78401764,  0.76763167,  0.74492137,  0.68605368,
        0.64431717,  0.62988067,  0.58528366,  0.51019485,  0.49952481,
        0.49911344,  0.49768894,  0.36523882,  0.34676968,  0.2187577 ,
        0.14526652,  0.12440816,  0.06250762, -0.04931043, -0.13791412,
       -0.22589908, -0.25058425, -0.28619054, -0.4569269 , -0.59463105,
       -0.60168851, -0.67789102, -0.6834685 , -0.68655654, -0.71258052,
       -0.8092249 , -0.8106505 , -0.81074737, -0.81703201, -0.81797771,
       -0.81961536, -0.82021801, -0.81949062, -0.81862127, -0.81720602,
       -0.81458848, -0.81335573, -0.80578976, -0.78834885, -0.77641598,
       -0.7757134 , -0.7388891 , -0.68442107, -0.63014655, -0.49875293,
       -0.48721433, -0.47345466, -0.44524657, -0.3967345 , -0.34802731,
       -0.31644253, -0.23374568, -0.23018377, -0.22420457, -0.22130633])
```

In [729…
```
u2 = fit_direct(x_test_data,y_test_data,0.0001)
yTest_pred =  predict_direct(x_test_data,x_test_data,u2[1])
```

In [730…
```
yTest_pred
```

Out[730]:
```
array([ 1.58953824,  1.58553028,  1.58361456,  1.56464928,  1.5227927 ,
        1.50359145,  1.48612587,  1.45560686,  1.45538281,  1.42509722,
        1.33919862,  1.30443444,  1.14834483,  1.05872443,  1.01890114,
        0.89906448,  0.85132768,  0.83869846,  0.82850794,  0.73950848,
        0.72373396,  0.70791406,  0.62804084,  0.61812338,  0.38115703,
        0.3363546 ,  0.215594  ,  0.1745198 ,  0.15339395,  0.11852635,
        0.09612798,  0.07483337,  0.03309077,  0.01791159, -0.00701049,
       -0.01494488, -0.02160014, -0.27226194, -0.34447337, -0.35343647,
       -0.37920622, -0.66224089, -0.69440572, -0.7042175 , -0.73220383,
       -0.827105  , -0.8469515 , -0.85962656, -0.86539602, -0.86944195])
```

In [731…
```
rmse = kernel_rmse(y_test_data,yTest_pred)
```

In [732…
```
rmse
```

Out[732]:
```
0.2954617474850962
```

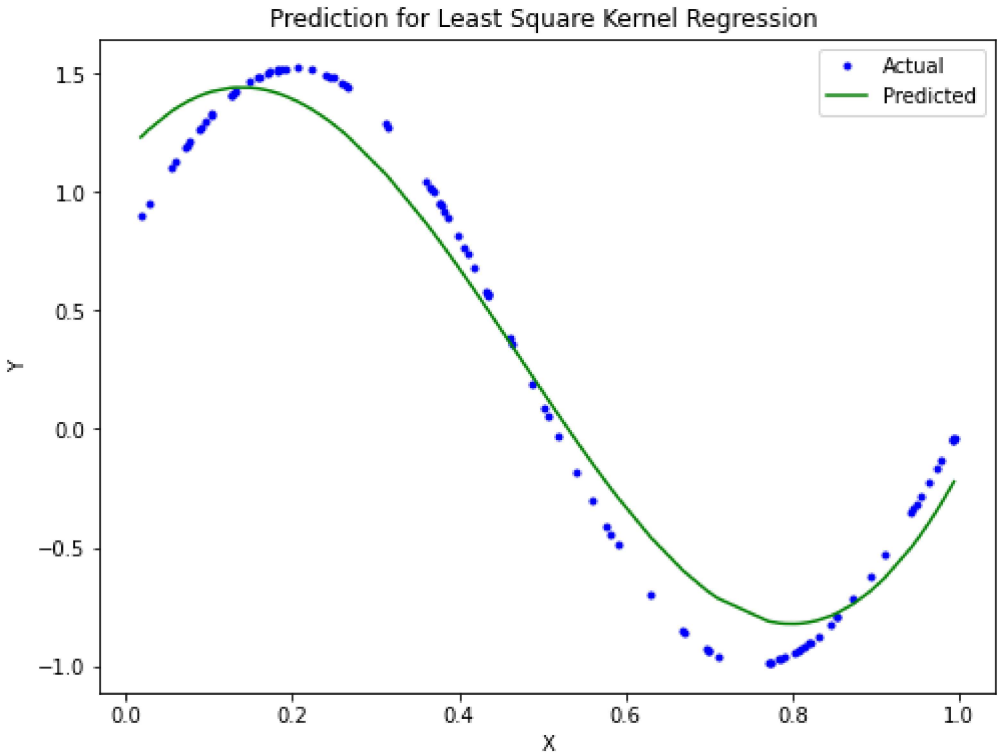
In [733…
```
mae = MAE(y_test_data, yTest_pred, 50)
```

```
mae
```

Out[733]:
```
0.2612141546285917
```

In [734…
```
def pltGraph(x, y, y_pred):

    fig = plt.figure(figsize=(8,6))

    #plotting the actual values
    plt.plot(x, y, 'b.')

    #prediction
    xs, ys = zip(*sorted(zip(x, y_pred)))

    plt.plot(xs,ys,'g')
    plt.legend([ "Actual","Predicted"])
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.title('Prediction for Least Square Kernel Regression ')
    plt.show()
```

In [735…
```
pltGraph(x_train_data,y_train_data,yTrain_pred)
```



Estimate the regularized Least Squares Kernel Regression model by using the Gaussian (RBF) kernel with stochastic gradient descent method. Make use of the tuned value of regularization parameter λ and kernel parameter σ of question 6. Compare the obtained solution with the solution obtained at question 6.

In [736…

```
class Generate_dataset:

  def getTrainSet(self, m, n):
    # sort will help to plot data properly
    X = np.sort(np.random.uniform(0.0, 1.0, (m*n))).reshape(m, n)
    return X

  def getTestSet(self, X):
    m = X.shape[0]
    ep = np.random.normal(loc=0.0, scale =0.25, size =m) # eplsilon
    y = np.array([np.sin(2*np.pi*np.linalg.norm(X[i]))+ ep[i] for i in range(m)]).reshape(m,1)

    return y
```

In [737…

```
class SVR:
  def rbf_kernal_fun(self, x,y):
    sigma= 1.2
    K = np.exp(-(1 / (sigma**2)) * np.sum((x-y)**2))
    #K = np.exp(-(1 / (sigma**2)) * np.linalg.norm(((x-y))**2))
    return K

  def construct_kernel_matrix(self, x1, x2):

    # applying kernel fun to each combinations of test set samples
    K = np.zeros((x1.shape[0],x2.shape[0]))
    for i in range(x1.shape[0]):
      for j in range(x2.shape[0]):
        K[i,j] = self.rbf_kernal_fun(x1[i,:], x2[j,:])

    # K : n x n
    return K

  def fit_direct(self, X,y, lamda):
    #x1 = X[:, 0].reshape((X.shape[0],1))
    #x2 = X[:, 1].reshape((X.shape[0],1))
    K = self.construct_kernel_matrix(X,X)
    I = np.identity(X.shape[0]+1) #np.eye(np.size( K , 1))
    #add col at end bias
    K = np.insert(K, -1, 1, axis=1)
    theta = np.linalg.pinv( (K.T @ K) + (lamda * I )) @ (K.T @ y)
    return K, theta

  def predict_direct(self, xTest, xTrain, theta):

    K = self.construct_kernel_matrix(xTest,xTrain)
    # for bias insert 1 extra col of 1s at last col
    K = np.insert(K, -1, 1, axis=1)

    Y_pred = K @ theta

    return Y_pred


  def find_rmse(self, y, y_pred):
    x = np.sqrt(np.mean((y_pred - y)**2))
    return x


  def pltGraph(self, X, y, y_pred):
```

```
    x1 = X[:, 0].reshape((X.shape[0],1))
    x2 = X[:, 1].reshape((X.shape[0],1))

    ax = plt.axes(projection = '3d')
    ax.scatter3D(x1,x2,y, 'y')
    #ax.plot3D(x1, x2,y_pred.flatten() ,'g')

    ax.set_xlabel('x1')
    ax.set_ylabel('X2')
    ax.set_zlabel('Y')
```

In [737…

In [738…

```
def rbf_kernel(x,y):
    sigma=1.2
    K = np.exp(-(1/(sigma**2)) *  np.sum((x-y)**2))
    return K

def kernel_matrix(x1, x2):
    K = np.zeros((x1.shape[0], x2.shape[0]))
    for i in range(x1.shape[0]):
        for j in range(x2.shape[0]):
            K[i,j] = rbf_kernel(x1[i], x2[j])
    return K

def fit_direct(x,y,l):
    K = kernel_matrix(x,x)
    I = np.identity(x.shape[0]+1)

    K = np.insert(K,-1,1, axis=1)

    theta = np.linalg.pinv((K.T @ K) + (l * I)) @ (K.T @ y)

    return K, theta
```

In [739…

```
data = Generate_dataset()
xTrain = data.getTrainSet(400, 2)
yTrain = data.getTestSet(xTrain)

xTest = data.getTrainSet(200,2)
yTest = data.getTestSet(xTest)

def fit_direct(self, X,y, lamda):
  my = SVR()
```

```
  K, theta = my.fit_direct(xTrain, yTrain, 0.005)
  y_pred = my.predict_direct(xTest, xTrain, theta)
  x=my.find_rmse(yTest,y_pred)
  print("rmse: ", x)
```

In [740…

```
def RMSE(y, y_pred):
    rmse = np.sqrt(np.mean((y_pred - y)**2))
    return rmse

def gradients(X, y, y_hat,lamda,w):


    m = X.shape[0]

    dw = (1/m)*np.dot(X.T, (y_hat - y)) + lamda * w
    db = (1/m)*np.sum((y_hat - y))

    return dw, db

def x_transform(X, degrees):


    X = np.array(X)
    ip = X.shape[1]
    nm = factorial(ip + degrees[-1])
    dn = factorial(degrees[-1]) * factorial(ip)
    nOP = int(nm / dn) - 1


    combinations = [combinations_with_replacement(range(ip),i) for i in range(1, degrees[-1] + 1)]
    combinations = [item for sublist in combinations for item in sublist]
    x_tr = np.empty((X.shape[0],nOP))

    for i, idx in enumerate(combinations):
        x_tr[:, i] = np.prod(X[:, idx], axis=1)

    return x_tr



def train_pol(X, y, bs, degrees, epochs, lr):

    if degrees[0]!=1:
      x = x_transform(X, degrees)
    else:
```

```
      x=X

    # m : no of samples , n : no of features
    m, n = x.shape


    w = np.zeros((n,1))
    b = 0

    lamda=0.0000002


    y = y.reshape(m,1)

    rmse = []


    for epoch in range(epochs):
        for i in range((m-1)//bs + 1):
            start_i = i*bs
            end_i = start_i + bs
            xb, yb = x[start_i:end_i], y[start_i:end_i]

            y_hat = np.dot(xb, w) + b

            dw, db = gradients(xb, yb, y_hat,lamda,w)

            w -= lr*dw
            b -= lr*db

        l = RMSE(y, (x @ w) + b)
        rmse.append(l)

    # returning weights, bias and losses(List).
    return w, b, rmse


def predict(X, w, b, degrees):


    if degrees[0]!=1:
      x1 = x_transform(X, degrees)
    else:
      x1=X
```

```
    # Returning predictions.
    return np.dot(x1, w) + b
```

In [741…

```
def RMSE2(y, y_pred,v):
    rmse = np.sqrt(np.mean((y_pred - y)**2)) + ((1/2) * (v.T @ v))
    return rmse


def train_KR(X, y, bs, epochs, lr):
  my=SVR()
  x = my.construct_kernel_matrix(xTrain,xTrain)
  m, n = x.shape

  v = np.zeros((m,1))
  lamda=0.0000001
  y = y.reshape(m,1)
  rmse = []

  for epoch in range(epochs):
      for i in range((m-1)//bs + 1):

          start_i = i*bs
          end_i = start_i + bs
          xb, yb = x[start_i:end_i, :], y[start_i:end_i, :]

          y_hat = (lamda * (v.T @ v)) - (yb- (xb@v))
          dv = (lamda * v) - (xb.T @ (yb- (xb@v)))

          v -= lr*dv

      l = RMSE2(y, np.dot(x, v),v)
      rmse.append(l)

  return rmse

# Predicting function.
def predict(xTest, xTrain, v):
  my=SVR()
  X = my.construct_kernel_matrix(xTest,xTrain)
  return X @ v
```

Modify the training set T by picking up randomly 5 data points from the training set T and scale their yi values by 20. Now, find the estimate of Least Squares Polynomial Regression model of order M= 9 with modified training set. Plot the estimated function along with data points of modified training set. Also, write down your observations.

In [742…
```
y_scaled = y_train_data.copy()
y_scaled[0]*=20
y_scaled[20]*=20
y_scaled[40]*=20
y_scaled[50]*=20
y_scaled[80]*=20
```

In [743…
```
u = fit_direct(x_train_data,y_scaled,[0.0001])
y_scaledPred = predict_direct(x_train_data,x_train_data,u[1])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-743-85c6435335bf> in <module>
----> 1 u = fit_direct(x_train_data,y_scaled,[0.0001])
      2 y_scaledPred = predict_direct(x_train_data,x_train_data,u[1])

TypeError: fit_direct() missing 1 required positional argument: 'lamda'
```

In [ ]:
```
pltGraph(x_train_data,y_scaled,y_scaledPred)
```

In [ ]:
```
rmse =  kernel_rmse(y_scaled,y_scaledPred)
rmse
```

There is presence of outlier becaus eof which rmse value is high