RegressionProject

February 10, 2020

1 Regression Project

Team: Mourya Gangaraju, Vishal Devnale, Vasishtha Sohani, Evan Lucas

Table of Contents

1.1 Introduction

[1]:
```
#import the numpy library for matrix calculation
import numpy as np

#for data plotting
import matplotlib.pyplot as plt

# Fitting Polynomial Regression to the dataset
from sklearn.preprocessing import PolynomialFeatures
from sklearn.kernel_approximation import RBFSampler

import pandas as pd

import random
```

1.2 Supporting functions

Mean Square Error Calculation, Add Bias Term, Normalize - Denormalize feature matrix

1.2.1 Mean Square Error - mseCalc(tMeas,tPred)

Function that returns mean square error for two arrays of inputs

Usage:

mse = mseCalc(tMeas,tPred)

Inputs:

tMeas: Array of measured values

tPred: Array of predicted values

Outputs:

mse: Mean square error value for entire array

[2]:
```
def mseCalc(tMeas, tPred):
    mse = np.mean(np.square(tMeas - tPred), axis=0)
    return mse
```

1.2.2 Matrix Normalziation - normalizeMatrix(x,*extraArgs)

Gaussian normalization function. Normalizes arrays by each columns mean and standard deviation.

Usage:

xOut, mean, standardDeviation = normalizeMatrix(x) - determines mean and standard deviation based on data. Data is assumed to be in the form of points x features (rows are points, columns are features)

xOut, mean, standardDeviation = normalizeMatrix(x,extraArgs) - allows user to specify mean and standard deviation. This is useful for allowing user to use the same normalization between a training and test data set.

Inputs:

x: Matrix to normalize

*extraArgs: Optional. Mean, stdev stored as an array with row 0 being means and row 1 being standard deviations. This will use these statistics instead of calculating them based on the input data

Outputs:

xOut: Normalized matrix

me: Mean used for normalization

stdev: Standard deviation used for normalization

[3]:
```
def normalizeMatrix(x, *extraArgs):
    if len(extraArgs) > 0:
        statsArray = np.squeeze(np.array(extraArgs))
        me = statsArray[0, :]
        stdev = statsArray[1, :]
    else:
        me = np.mean(x, axis=0)
        stdev = x.std(axis=0)
        #print("Means - {} \nStandard Deviations - {} ".format(me, stdev))

    xOut = (x - me) / stdev
    return xOut, me, stdev
```

1.2.3 Matrix Denormalziation - deNormalizeMatrix(x,me,stdev)

Function to map data back into original units

Usage:

xScaled = deNormalizeMatrix(x,me,stdev)

Inputs:

x: Matrix to normalize

me: Mean

stdev: Standard deviation

Outputs:

xScaled: Denormalized matrix

[4]:
```
def deNormalizeMatrix(x, me, stdev):
    xScaled = x * stdev + me
    return xScaled
```

1.2.4 Add Bias Term - addBiasTerm(x)

Function adds bias term (x0=1) for all data samples

Usage:

y = addBiasTerm(x)

Input:

x: Matrix to be biased

Output:

y: Matrix with column of 1s inserted in first column

[5]:
```
def addBiasTerm(x):
    y = np.insert(x, 0, 1, axis=1)
    return y
```

1.2.5 Split data into training and test - my_train_test_split(ldata, lpercentage, lrandoma)

Function splits all data samples into training and test data. It randomizes if lrandoma is set to True.

Usage:

train_data, test_data = my_train_test_split(data, 20, False) train_data, test_data = my_train_test_split(data, 10, True)

Input:

ldata: data to be splitted into train and test

lpercentage: percentage of test data

lrandoma: randomizes input data if lrandoma is set to True

Output:

ltrain: train data

ltest: test data

```
[6]: def my_train_test_split(ldata, lpercentage, lrandoma):
         l_training_data_length = len(ldata[:, 0]) - int(
             len(ldata[:, 1]) / lpercentage)
         l_test_data_length = int(len(ldata[:, 0]) / lpercentage)
         if lrandoma == True:
             random.shuffle(ldata)

         ltrain = ldata[:l_training_data_length, :]
         ltest = ldata[l_training_data_length:, :]

         #error detection code
         if (ltrain.shape[0] + ltest.shape[0] != ldata.shape[0]):
             print(
                 'Error: Summation of length(rows) of test and train data is NOT␣
       ↪equal to input data'
             )
             print('Input Data length - {}, Test - {}, Train - {}'.format(
                 ldata.shape[0], ltest.shape, ltrain.shape[0]))

         return ltrain, ltest
```

1.3 Regression function definitions

1.3.1 Linear Regression Function

Basic linear regression function. It is suggested that a column of 1s is added to the feature matrix before running this function so that there can be a linear bias term.

Usage:

weight_matrix = linregression(feature_matrix,output_matrix)

Inputs:

feature_matrix: array of features. Data is assumed to be in the form of points x features (rows are points, columns are features)

output_matrix: Target/output values. Data is assumed to be in the form of points x features (rows are points, columns are features)

Output:

weight_matrix: Array of weights connecting output and features. Eg. output = w_0*1+w_1*input1…+w_n*inputN

```
[7]: def linregression(feature_matrix, output_matrix):
         weight_matrix = np.matmul(np.matmul(np.linalg.pinv(np.matmul(feature_matrix.
       ↪T,feature_matrix)),\
                                             feature_matrix.T),output_matrix)
         return weight_matrix
```

1.3.2 Regularized Linear Regression

Basic linear regression function with L2 regularization to reduce overfitting errors. It is suggested that a column of 1s is added to the feature matrix before running this function so that there can be a linear bias term.

Usage: weight__matrix = linregression(feature__matrix,output__matrix,lam)

Inputs:

feature__matrix: array of features. Data is assumed to be in the form of points x features (rows are points, columns are features)

output__matrix: Target/output values. Data is assumed to be in the form of points x features (rows are points, columns are features)

lam: Regularization constant

Output:

weight__matrix: Array of weights connecting output and features. Eg. output = w__0*1+w__1*input1…+w__n*inputN

[8]:
```python
def linregressionRegularized(feature_matrix, output_matrix, lam):
    weight_matrix = np.matmul(
        np.linalg.pinv(
            np.matmul(feature_matrix, feature_matrix.T) +
            lam * np.eye(len(feature_matrix))), feature_matrix)
    weight_matrix = np.matmul(weight_matrix.T, output_matrix)
    return weight_matrix
```

1.3.3 Polynomial Linear Regression

Basic polynomial regression function. It is suggested that a column of 1s is added to the feature matrix before running this function so that there can be a linear bias term.

Usage: weight__matrix = linregressionPolynomial(feature__matrix,output__matrix,ldegree)

Inputs:

feature__matrix: array of features. Data is assumed to be in the form of points x features (rows are points, columns are features)

output__matrix: Target/output values. Data is assumed to be in the form of points x features (rows are points, columns are features)

ldegree: degree of polynomial to calculate weight

lambda: Regularization constant

Output:

weight__matrix: Array of weights connecting output and features. Eg. output = w__0*1+w__1*input1…+w__n*inputN

```
[9]: def linregressionPolynomial(feature_matrix, output_matrix, ldegree, llam):
         poly = PolynomialFeatures(degree=ldegree)
         poly_feature = poly.fit_transform(feature_matrix)
         weight_matrix = linregressionRegularized(poly_feature, output_matrix, llam)

         lmse = mseCalc(np.matmul(poly_feature, weight_matrix), output_matrix)

         return weight_matrix, lmse
```

1.4 Requested outputs

1.4.1 Question 1 work

Three flavors of my_regression were created.

my_regression_1D - original version of my_regression, but not well suited for multiple outputs. Varies polynomial degree and regularization constant

my_regressionRBF - Modified version of my_regression_1D that transforms features into RBFs and performs linear regression on them. Ideally this would have been merged, but that seemed tedious and we had already accomplished the assignment requirements. Varies number of basis functions and gamma (which sets width of the basis functions). RBFSampler doesn't exactly do RBF sampling, but it creates a very close approximation - see sklearn documentation for details on this.

my_regression - Final version. Calls my_regression_1D after splitting outputs.

For all of these, the function definition is below:

def my_regression(trainX, testX, noutputs)

def my_regression(trainX, testX, noutputs)

The input variables are:

```
trainX - an [ntrain x (nfeature + noutputs)] array that contains the features in the first 'nfe
testX - an [ntest x nfeature] array of test data for which the predictions are made
noutputs - the number of output columns in trainX
```

The output should be an [ntest x noutputs] array, which contains the prediction values for the testX data. You can use these data to then calculate squared error by comparing against the testX outputs.

Your my_regression code should do some kind of cross-validation to determine the right model for the training data, e.g., linear vs. polynomial vs. radial basis functions (your choice). Then this model is applied to the test data to make a prediction.

```
[10]: def my_regression_1D(trainX, testX, numOfOutputs):

          #error detection code
          if (trainX.shape[1] - testX.shape[1]) != numOfOutputs:
              print('Error - trainX({}), testX({}), numOfOutputs({}), \
                    does NOT match the function requirement'.format(
        trainX.shape[1], testX.shape[1], numOfOutputs))

#Create Folds - Split data into '5' folds
nfolds = 5

#number of samples provided for training
ntrain = trainX.shape[0]
indices = np.linspace(0, ntrain, nfolds + 1, dtype=int)

degree = [1, 2, 3, 4, 5]
lamd = [0.01, 0.1, 1, 10, 50, 100]

errordata2 = np.zeros((len(indices) - 1, len(degree), len(lamd)))
xaverage = np.zeros((len(lamd), len(degree)))

for ii in range(len(indices) - 1):
    test_data = trainX[indices[ii]:indices[ii + 1]]
    train_data = trainX
    train_data = np.delete(train_data, np.s_[indices[ii]:indices[ii + 1]],0)

    #Taking feature matrix and output matrix from different folds
    train_data_ftr = train_data[:, :-numOfOutputs]
    train_data_ftr = addBiasTerm(train_data_ftr)

    train_data_out = train_data[:, -numOfOutputs]

    #Taking Feature Matrix and output matrix from test data
    test_data_ftr = test_data[:, :-numOfOutputs]
    test_data_ftr = addBiasTerm(test_data_ftr)

    test_data_out = test_data[:, -numOfOutputs]

    #polynomial fit of the feature matrix and test matrix
    errordata = []

    for iii in range(len(degree)):
        i = degree[iii]
        polyftr = PolynomialFeatures(degree=i)
        polyfit_ftr_matrix = polyftr.fit_transform(train_data_ftr)
        polyfit_testftr_matrix = polyftr.fit_transform(test_data_ftr)
        for jjj in range(len(lamd)):
            j = lamd[jjj]
            weight_matrix = linregressionRegularized(
                polyfit_ftr_matrix, train_data_out, j)
            test_predicted = np.matmul(polyfit_testftr_matrix,
                                       weight_matrix)
            error = mseCalc(test_data_out, test_predicted)
```

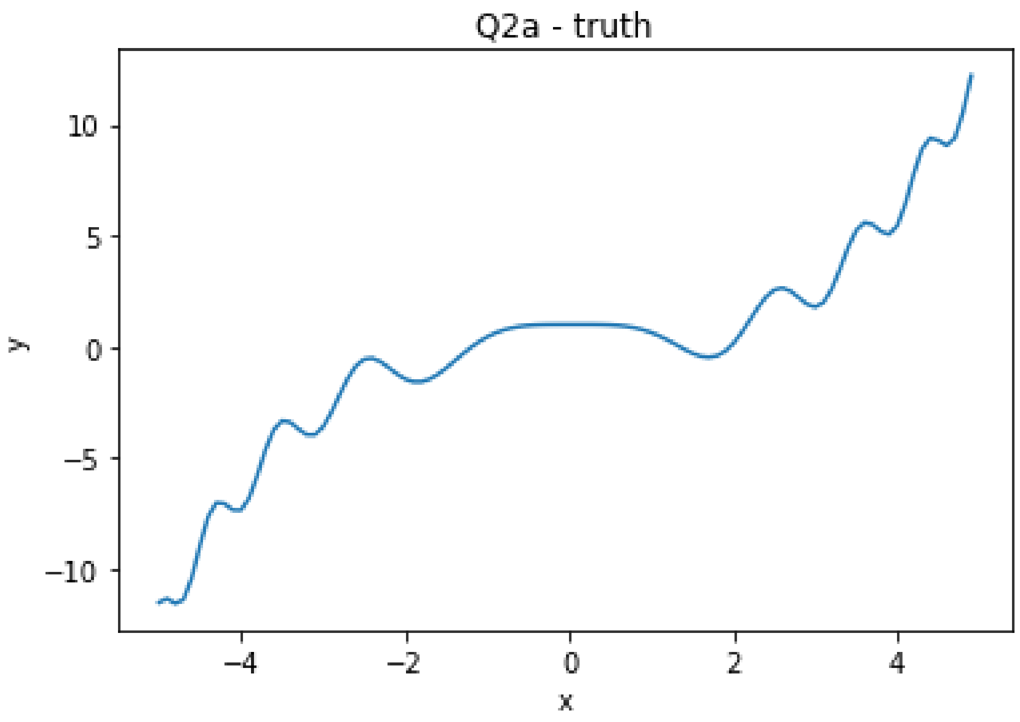
```
                errordata.append([ii, i, j, error])
                errordata2[ii, iii, jjj] = error

        df = pd.DataFrame(columns=["Fold", "Degree", "lambda", "error"],
                          data=errordata)

        if printDf == True:
            print(df)

    xaverage = np.mean(errordata2,axis=0)

    p = np.unravel_index(xaverage.argmin(), xaverage.shape)

    #print("Minimum MSE is at degree = {} and lambda = {}".format(degree[p[0]],␣
↪lamd[p[1]]))

    #now let's train entire data with degree (degree[p[0]]) and lambda␣
↪(lamd[p[1]]) found in CV
    train_feature_matrix = addBiasTerm(trainX[:, :-numOfOutputs])
    train_output_matrix = trainX[:, -numOfOutputs]

    polyftr = PolynomialFeatures(degree=degree[p[0]])
    polyfit_ftr_matrix = polyftr.fit_transform(train_feature_matrix)
    weight_matrix = linregressionRegularized(polyfit_ftr_matrix,␣
↪train_output_matrix, lamd[p[1]])
    test_predicted = np.matmul(polyfit_ftr_matrix, weight_matrix)
    xmse = mseCalc(train_output_matrix, test_predicted)

    polyftr = PolynomialFeatures(degree=degree[p[0]])
    polyfit_ftr_matrix = polyftr.fit_transform(addBiasTerm(testX))
    test_predicted = np.matmul(polyfit_ftr_matrix, weight_matrix)

    return test_predicted
```

[11]:
```
def my_regression(trainX, testX, pOutputs):

    output = np.zeros((testX.shape[0],1))

    #check the number of outputs and call my_regression_1D
    for k in range(pOutputs):
        trainXX = np.concatenate((trainX[:,:-pOutputs], np.asmatrix(trainX[:
↪,-(pOutputs-k)]).T), axis=1)
        output = np.concatenate((output, my_regression_1D(trainXX, testX, 1)),␣
↪axis=1)

    #remove first column
    output = output[:,1:]
    return output
```

1.4.2 Question 2 work

2A Plot y versus x (y on vertical axis, x on horizontal axis) as a line with noise variance = 0. This is the truth that you wish your regression model to fit to. Plot from x = -5 to +5 at increments of 0.1.

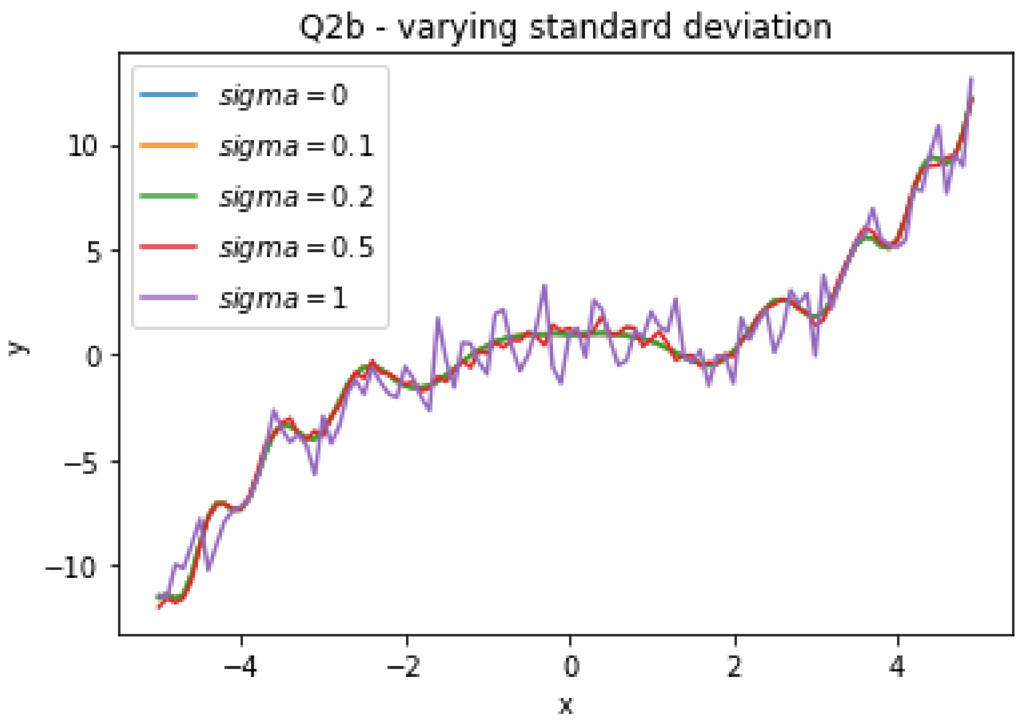
[12]:
```
xx = range(-50, 50, 1)
x = np.asarray(list(xx)) / 10

sigma = 0
epsilons = np.random.normal(0, sigma**2, 100)

ys = np.cos(x**2) + 0.1 * x**3 + epsilons

ax1 = plt.plot(x, ys)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Q2a - truth")
plt.show()
```



2B Plot y versus x for standard deviations sigma=0.1,0.2,0.5,1.

Each line should be a different color. Make sure your plots are labeled with a legend. You will now use your regression function to see how close you can come to the truth in part a) based on noisy measurements.

[13]:
```
sigmas = [0, 0.1, 0.2, 0.5, 1]

for sigma in sigmas:
    epsilons = np.random.normal(0, sigma**2, 100)
    ys2 = np.cos(x**2) + 0.1 * x**3 + epsilons
    ax2 = plt.plot(x, ys2, label='$sigma = {sigma}$'.format(sigma=sigma))
    mse = mseCalc(ys, ys2)
    #print(mse)

plt.xlabel("x")
plt.ylabel("y")
plt.title("Q2b - varying standard deviation")
plt.legend(loc='best')
plt.show()
```



2C Now you will run an experiment where you will see how close your regression is to the truth based on the variance of the noise and the number of training data. Make a table that has number of training data [2,5,10,20,50,100,200,500] in the rows and the standard deviations from part b) in the column. Run 5-fold cross validation on each of these 32 combinations of parameters and report the cross-validation squared error.

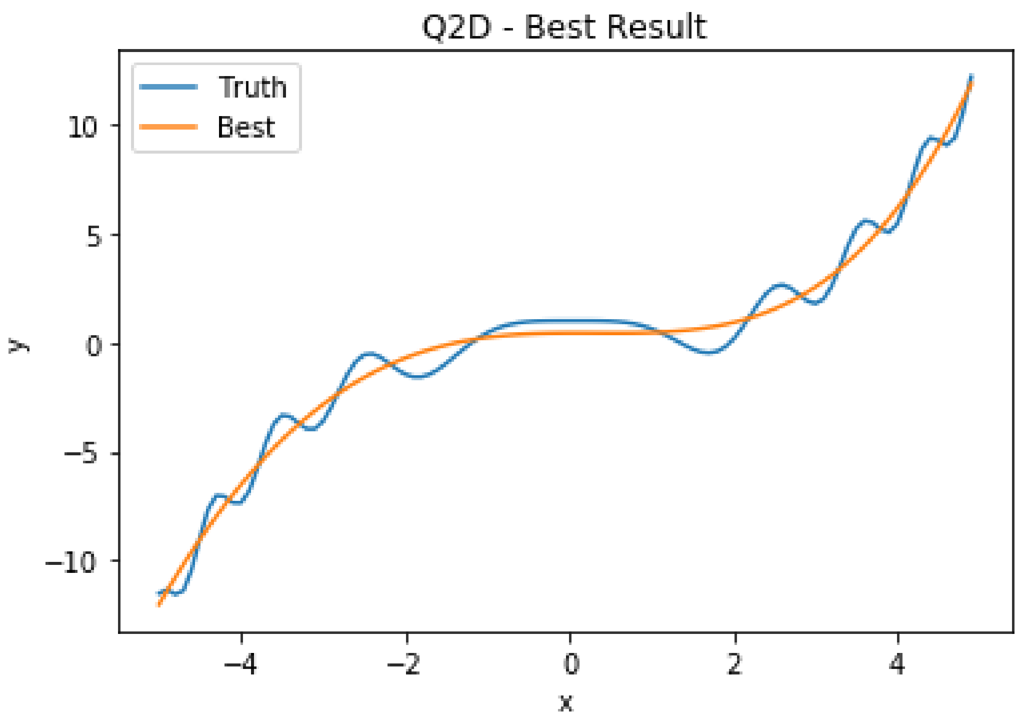
```
[23]: printDf = False

y_truth = np.asmatrix(ys).T
my_testX = np.asmatrix(x).T

sigmas = [0.1 , 0.2 , 0.5 , 1]
numOfSamples = [5,10,20,50,100,200,500]

MSE_MATRIX = np.zeros((len(numOfSamples), (len(sigmas))))
my_predicted_data_store = np.zeros((len(numOfSamples), len(sigmas),␣
 ↪len(y_truth)))

for m in range(len(numOfSamples)):
    numOfSample = numOfSamples[m]
    feature_sample = random.choices(x, k=numOfSample)
    feature_sample = np.asmatrix(feature_sample).T
    for p in range(len(sigmas)):
        sigma = sigmas[p]
        epsilons = np.random.normal(0, sigma**2, numOfSample)
        epsilons = np.asmatrix(epsilons).T
        output_x = np.cos(np.square(feature_sample)) + 0.1 * np.
 ↪power(feature_sample, 3) + epsilons
        my_trainX = np.concatenate((feature_sample, output_x), axis=1)
        my_predicted_data = my_regression_1D(my_trainX, my_testX, 1)
        my_predicted_data_store[m,p] = my_predicted_data.T
        MSE_MATRIX[m, p] = mseCalc(my_predicted_data, y_truth)

Q2C_MSE = pd.DataFrame(MSE_MATRIX, columns = ['Sigma=0.1' , 'Sigma=0.2' ,␣
 ↪'Sigma=0.5' , 'Sigma=1'], \
                                                                         ␣
 ↪index=['n=5','n=10','n=20','n=50','n=100','n=200','n=500'])
print(Q2C_MSE)
```

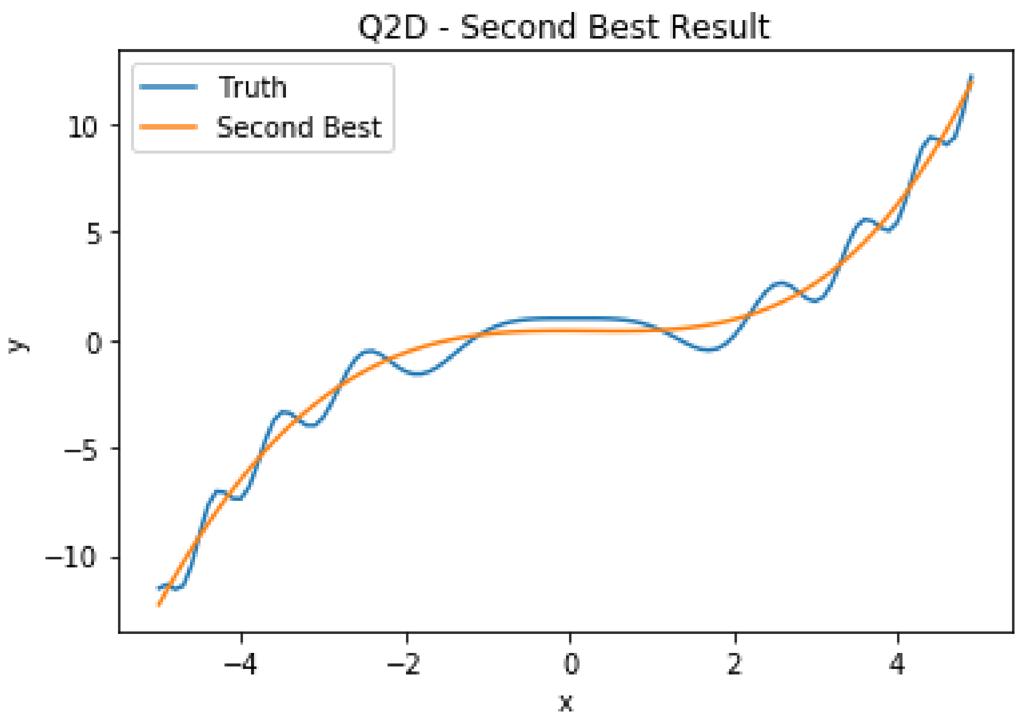
```
       Sigma=0.1  Sigma=0.2  Sigma=0.5    Sigma=1
n=5    12.708308  12.847686  11.332072  10.988469
n=10    1.050092   1.045976   1.682959  14.289919
n=20    0.579998   0.578776   0.572529   0.588628
n=50    0.499496   0.495653   0.509534   0.546492
n=100   0.488977   0.483976   0.487263   0.527849
n=200   0.478777   0.478079   0.479844   0.523865
n=500   0.479875   0.479267   0.481970   0.510797
```

2D Select the two best results (lowest error) and two worst results (highest error) from part c) and make four plots. Each plot should show the truth line you plotted in part a) and the learned regression model. Plot from x = -5 to 5 at increments of 0.1. What do you notice about these results?

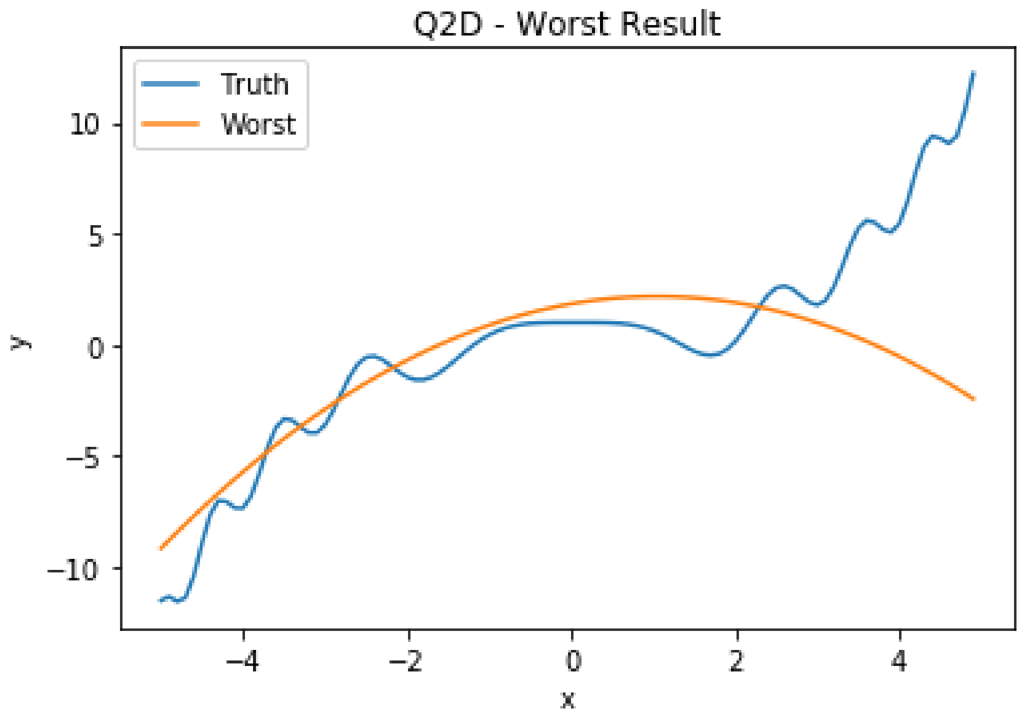
[24]:
```
#Best Result
px1 = np.unravel_index(MSE_MATRIX.argmin(), MSE_MATRIX.shape)
print('Min-1' ,px1, numOfSamples[px1[0]], sigmas[px1[1]])

ax21 = plt.plot(x, ys,label = 'Truth')
plt.plot(x, my_predicted_data_store[px1[0],px1[1]],label = 'Best')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Q2D - Best Result")
plt.legend(loc='best')
plt.show()

ax22 = plt.plot(x, ys,label = 'Truth')
plt.plot(x, my_predicted_data_store[6,1],label = 'Second Best')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Q2D - Second Best Result")
plt.legend(loc='best')
plt.show()

#Worst Result
px2 = np.unravel_index(MSE_MATRIX.argmax(), MSE_MATRIX.shape)
print('Max-1', px2, numOfSamples[px2[0]], sigmas[px2[1]], 'Max')

ax21 = plt.plot(x, ys,label = 'Truth')
plt.plot(x, my_predicted_data_store[px2[0],px2[1]],label = 'Worst')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Q2D - Worst Result")
plt.legend(loc='best')
plt.show()

ax22 = plt.plot(x, ys,label = 'Truth')
plt.plot(x, my_predicted_data_store[0,1],label = 'Second Worst')
plt.xlabel("x")
plt.ylabel("y")
plt.title("Q2D - Second Worst Result")
plt.legend(loc='best')
plt.show()
```

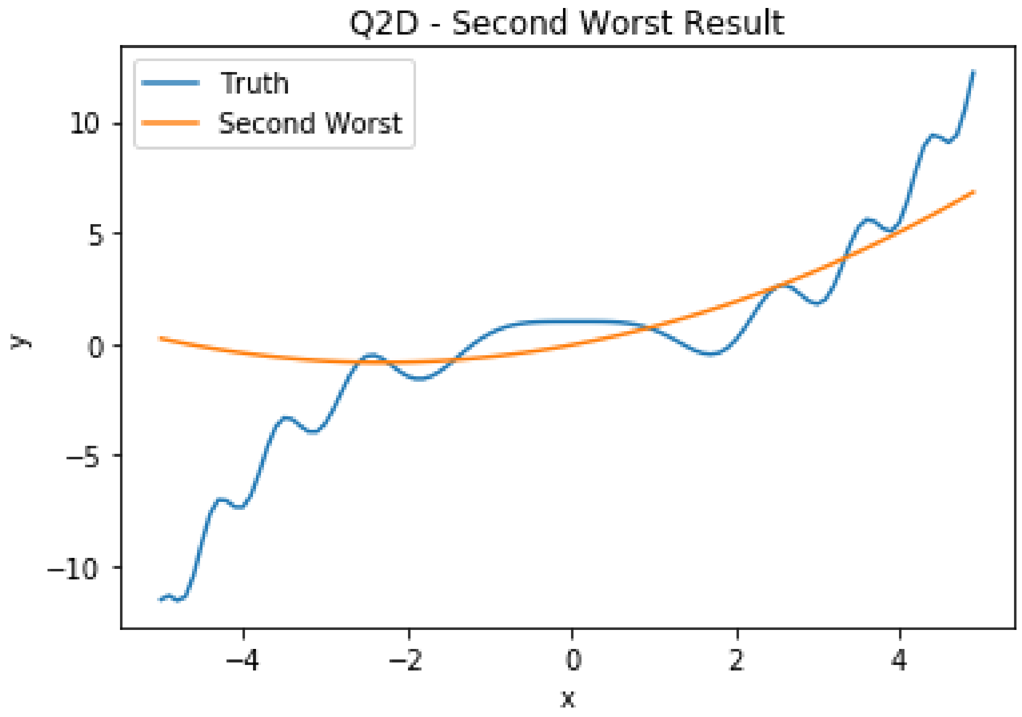
```
Min-1 (5, 1) 200 0.2
```


Q2D - Best Result
Truth
Best
10
5
0
−5
−10
y
−4
−2
0
2
4
x


Q2D - Second Best Result
Truth
Second Best
10
5
0
−5
−10
y
−4
−2
0
2
4
x

```
Max-1 (1, 3) 10 1 Max
```





Q2-D Discussion First, because we are randomly sampling the points, the fits that are best change everytime we run it.

Second, if we see a second or first order fit be selected, it is invariably the worst.

Third, we had some issues making our code work with 2 points - this would always calculate a linear model. Higher order models would

1.4.3 Question 3 work

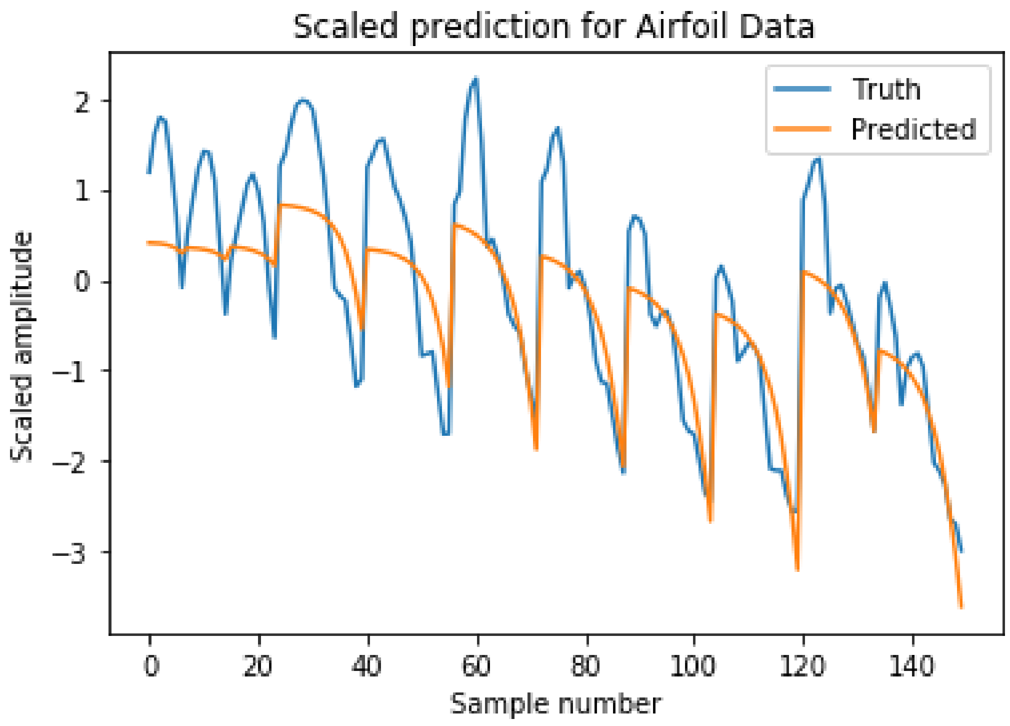
Now run your regression algorithm for the three UCI data sets above. Run five-fold cross-validation to estimate the squared error of your learned model for each data set. Comment on the results.

Airfoil Self-Noise

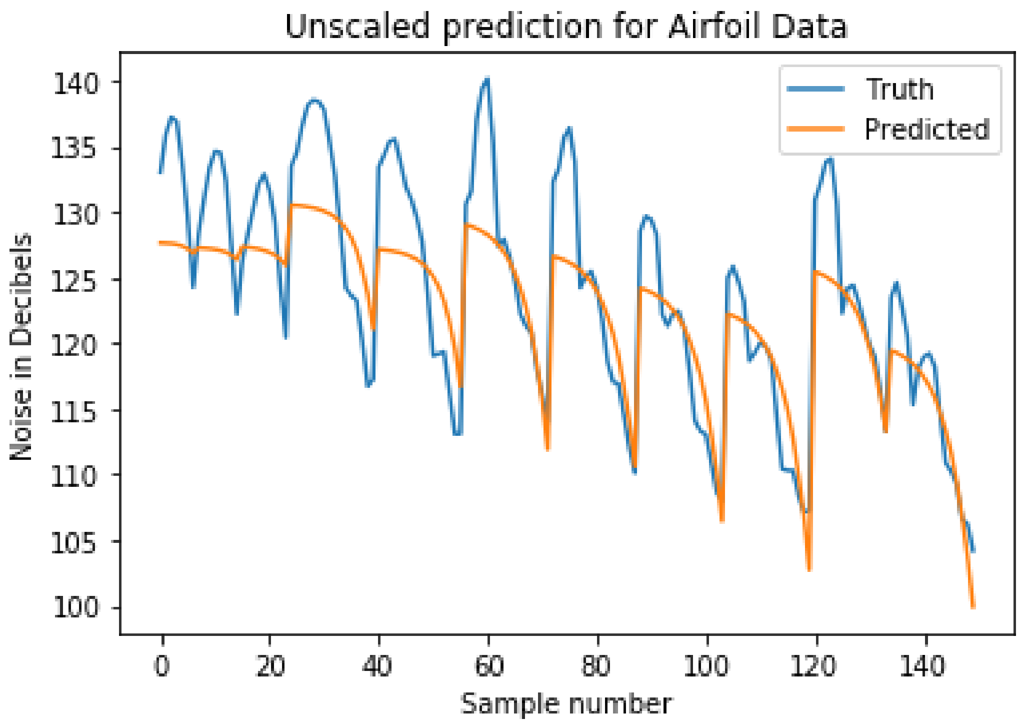
[16]:
```
#load data
airfoil_data = np.loadtxt("airfoil_self_noise.dat")

#this variable needs to change for each database
numOfOutputs = 1

#normalize data
airfoil_data, featureMeans, featureStdevs = normalizeMatrix(airfoil_data)

#split into train and test data
train_data, test_data = my_train_test_split(airfoil_data, 10, False)

#call my_regression
printDf = False
test_data_prediction = my_regression(train_data, test_data[:, :
 ↪-numOfOutputs],numOfOutputs)

#to see both output side-by-side
#print(np.concatenate((test_data[:,-numOfOutputs:], test_data_prediction),␣
 ↪axis=1))

#print MSE of test_data
print('Scaled MSE - {}'.format(
    mseCalc(test_data_prediction, test_data[:, -numOfOutputs:])))

ax4 = plt.plot(test_data[:, -numOfOutputs:], label="Truth")
plt.plot(test_data_prediction, label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("Scaled amplitude")
plt.title("Scaled prediction for Airfoil Data")
plt.legend(loc = 'best')
plt.show()
```

```
#Do it again unscaled because I think it looks better
unScaledPrediction =␣
 ↪deNormalizeMatrix(test_data_prediction,featureMeans[-numOfOutputs:],␣
 ↪featureStdevs[-numOfOutputs:])
unScaledTarget = deNormalizeMatrix(test_data[:, -numOfOutputs:
 ↪],featureMeans[-numOfOutputs:], featureStdevs[-numOfOutputs:])

#print MSE of test_data
print('Unscaled MSE - {}'.format(mseCalc(unScaledPrediction, unScaledTarget)))

ax44 = plt.plot(unScaledTarget, label="Truth")
plt.plot(unScaledPrediction, label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("Noise in Decibels")
plt.title("Unscaled prediction for Airfoil Data")
plt.legend(loc = 'best')
plt.show()
```

```
Scaled MSE - [[0.44671849]]
```



```
Unscaled MSE - [[21.24584174]]
```

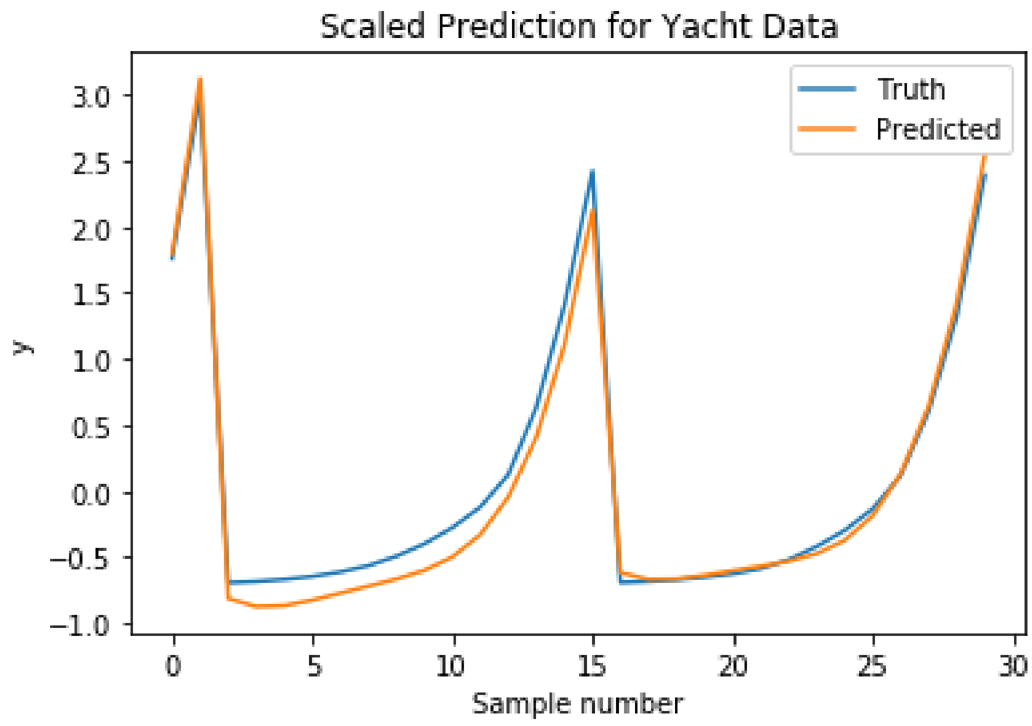


Yacht Hydrodynamics

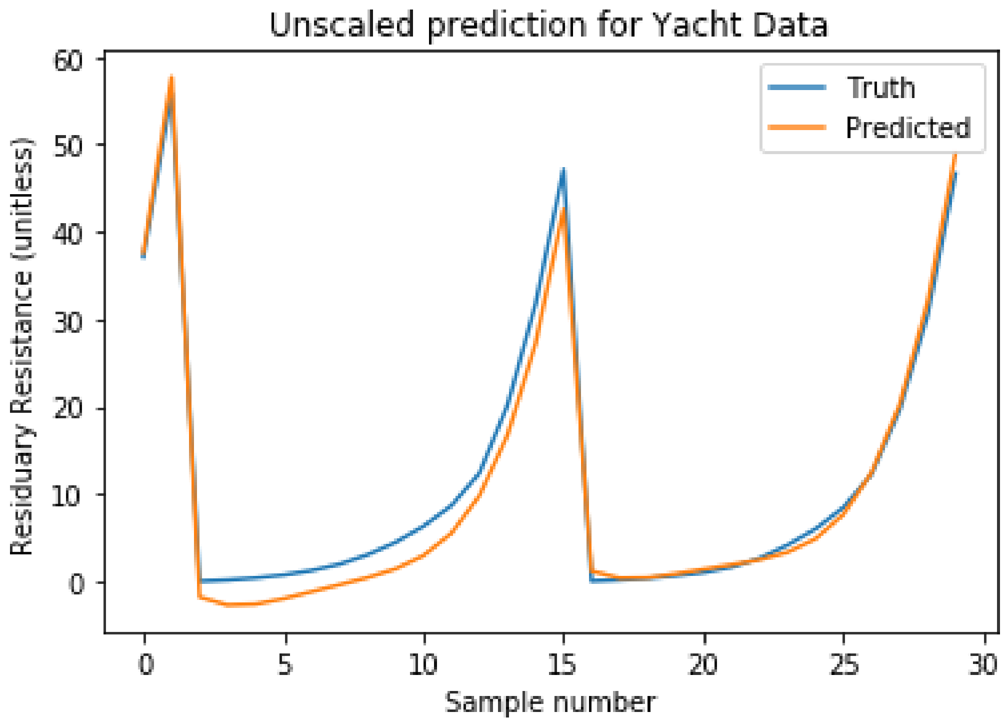
[17]:
```
#load data
yacht_hydrodynamics_data = np.loadtxt("yacht_hydrodynamics.data")

#this variable needs to change for each database
numOfOutputs = 1

#normalize data
yacht_hydrodynamics_data, featureMeans, featureStdevs =␣
 ↪normalizeMatrix(yacht_hydrodynamics_data)

#split into train and test data
train_data, test_data = my_train_test_split(yacht_hydrodynamics_data, 10, False)

#call my_regression
printDf = False
test_data_prediction = my_regression(train_data, test_data[:, :-numOfOutputs],␣
 ↪numOfOutputs)

#to see both output side-by-side
#print(np.concatenate((test_data[:,-numOfOutputs:], test_data_prediction),␣
 ↪axis=1))
```

```
#print MSE of test_data
print('MSE - {}'.format(mseCalc(test_data_prediction, test_data[:,␣
↪-numOfOutputs:])))

ax4 = plt.plot(test_data[:, -numOfOutputs:], label="Truth")
plt.plot(test_data_prediction, label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("y")
plt.title("Scaled Prediction for Yacht Data")
plt.legend(loc = 'best')
plt.show()

#Do it again unscaled because I think it looks better
unScaledPrediction =␣
↪deNormalizeMatrix(test_data_prediction,featureMeans[-numOfOutputs:],␣
↪featureStdevs[-numOfOutputs:])
unScaledTarget = deNormalizeMatrix(test_data[:, -numOfOutputs:
↪],featureMeans[-numOfOutputs:], featureStdevs[-numOfOutputs:])

#print MSE of test_data
print('Unscaled MSE - {}'.format(mseCalc(unScaledPrediction, unScaledTarget)))

ax44 = plt.plot(unScaledTarget, label="Truth")
plt.plot(unScaledPrediction, label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("Residuary Resistance (unitless)")
plt.title("Unscaled prediction for Yacht Data")
plt.legend(loc = 'best')
plt.show()
```

```
MSE - [[0.02218198]]
```



```
Unscaled MSE - [[5.08176265]]
```

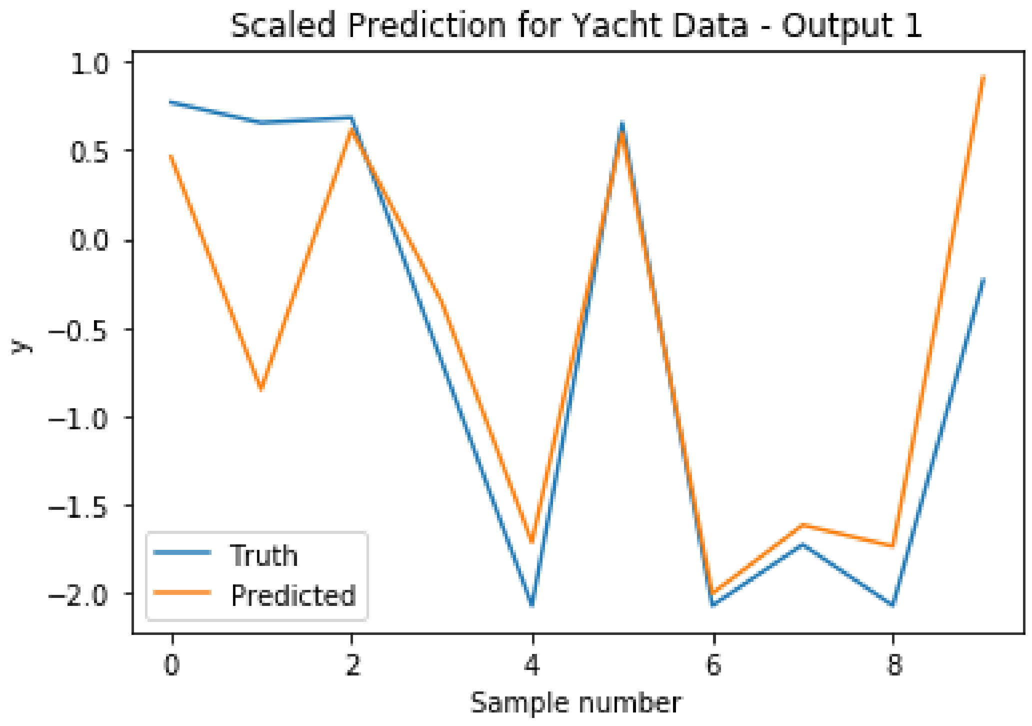


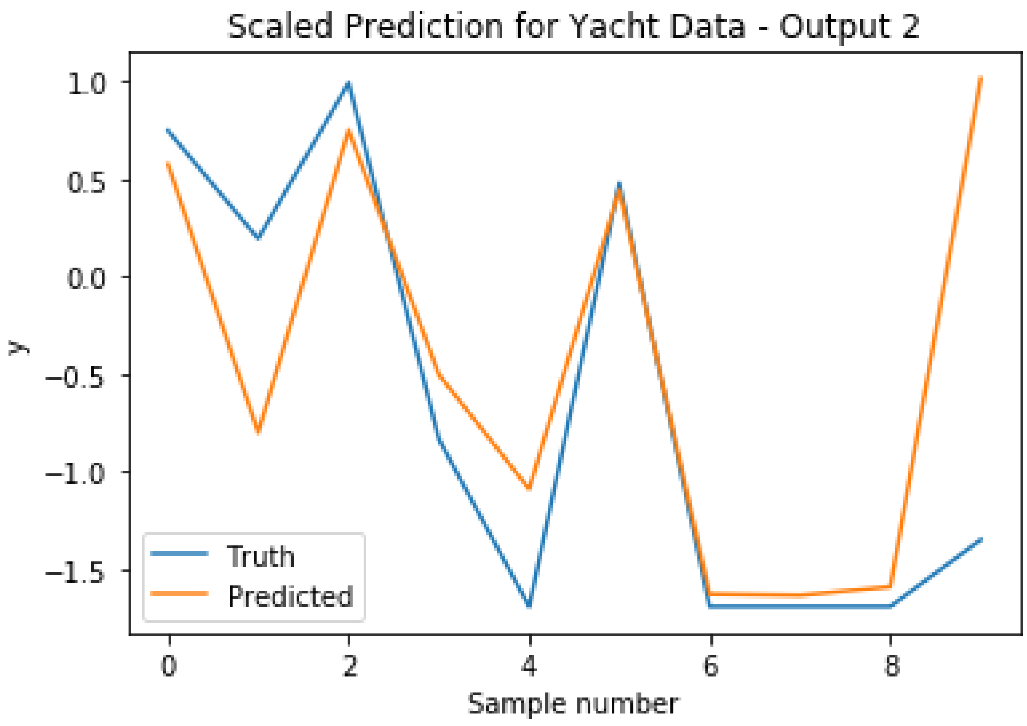
Concrete Slump

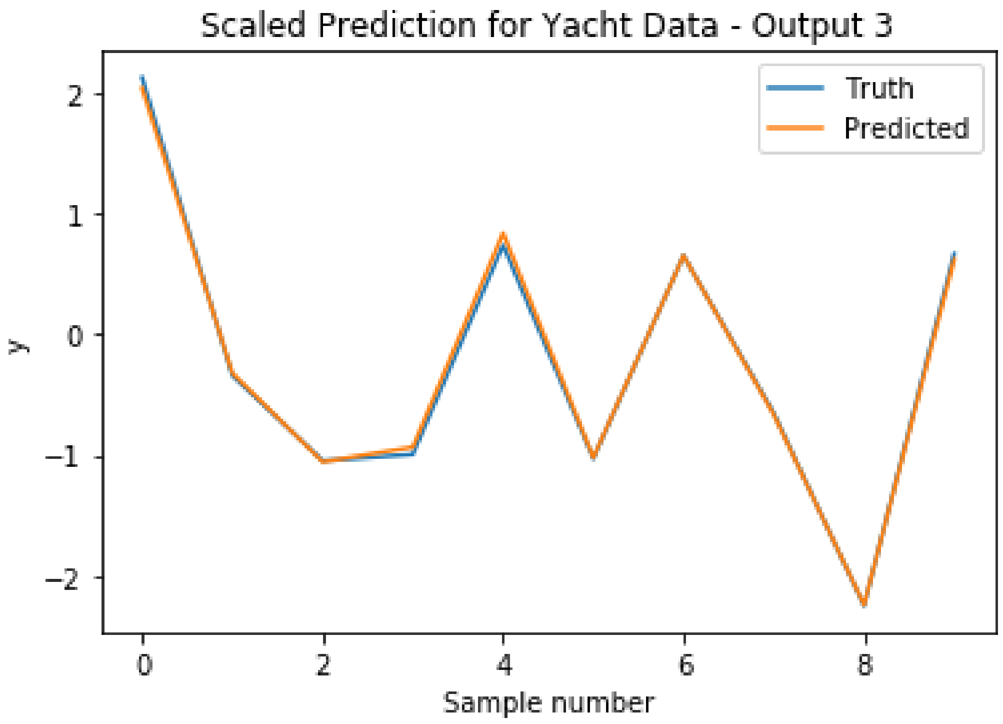
```
[29]: #load data
#concrete_data = np.loadtxt("slump_test.data",skiprows=1,delimiter=',')
concrete_data = np.loadtxt("slump_test.data",skiprows=1,usecols =␣
 ↪(1,2,3,4,5,6,7,8,9,10),delimiter=',')

#this variable needs to change for each database
numOfOutputs = 3

#normalize data
concrete_data, featureMeans, featureStdevs = normalizeMatrix(concrete_data)

#split into train and test data
train_data, test_data = my_train_test_split(concrete_data, 10, True)

#call my_regression
printDf = False
#print(train_data.shape, test_data[:, :-numOfOutputs].shape)
test_data_prediction = my_regression(train_data, test_data[:, :
 ↪-numOfOutputs],numOfOutputs)

#to see both output side-by-side
#print(np.concatenate((test_data[:,-numOfOutputs:], test_data_prediction),␣
 ↪axis=1))

#print MSE of test_data
print('MSE - {}'.format(mseCalc(test_data_prediction, test_data[:,␣
 ↪-numOfOutputs:])))

ax4 = plt.plot(test_data[:, -3], label="Truth")
plt.plot(test_data_prediction[:,0], label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("y")
plt.title("Scaled Prediction for Yacht Data - Output 1")
plt.legend(loc = 'best')
plt.show()

ax4 = plt.plot(test_data[:, -2], label="Truth")
plt.plot(test_data_prediction[:,1], label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("y")
plt.title("Scaled Prediction for Yacht Data - Output 2")
plt.legend(loc = 'best')
plt.show()
```

```python
ax4 = plt.plot(test_data[:, -1], label="Truth")
plt.plot(test_data_prediction[:,2], label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("y")
plt.title("Scaled Prediction for Yacht Data - Output 3")
plt.legend(loc = 'best')
plt.show()
```

```
MSE - [[0.40557565 0.71702564 0.0024033 ]]
```




Scaled Prediction for Yacht Data - Output 2
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0
2
4
6
8
Sample number
Truth
Predicted


Scaled Prediction for Yacht Data - Output 3
2
1
0
−1
−2
y
0
2
4
6
8
Sample number
Truth
Predicted

Discussion of results for Question 3 Not unsurprisingly, linear models were never selected as the best model for any of the real data sets.

Results we found with linear regression were in the same ballpark as squared errors released on Canvas - which helped build confidence before we moved onto polynomial regression.

The linear regression with a radial basis function transform had errors close to the polynomial regression output, but generally a well-fit polynomial regression worked better. One surprise with the radial basis functions was that the models using lower numbers of basis functions had lower test error during cross validation. The team assumed that a higher order model would offer a better fit, but this was not always the case. In general, low numbers of basis functions offered low training error with high testing error, so the lower limit of the number of basis functions was brought up to 30

The polynomial basis functions worked very well. Having code to transform the attributes into polynomial space was really nice to have. It was pretty wild to watch the code select both a degree of polynomial and a regularization constant on it's own.

1.5 Radial Basis Function

We have implemented Radial basis function and tried it on 'airfoil_self_noise.dat' and 'yacht_hydrodynamics.data'.

Note: Radial basis function is not considered in my_regression function.

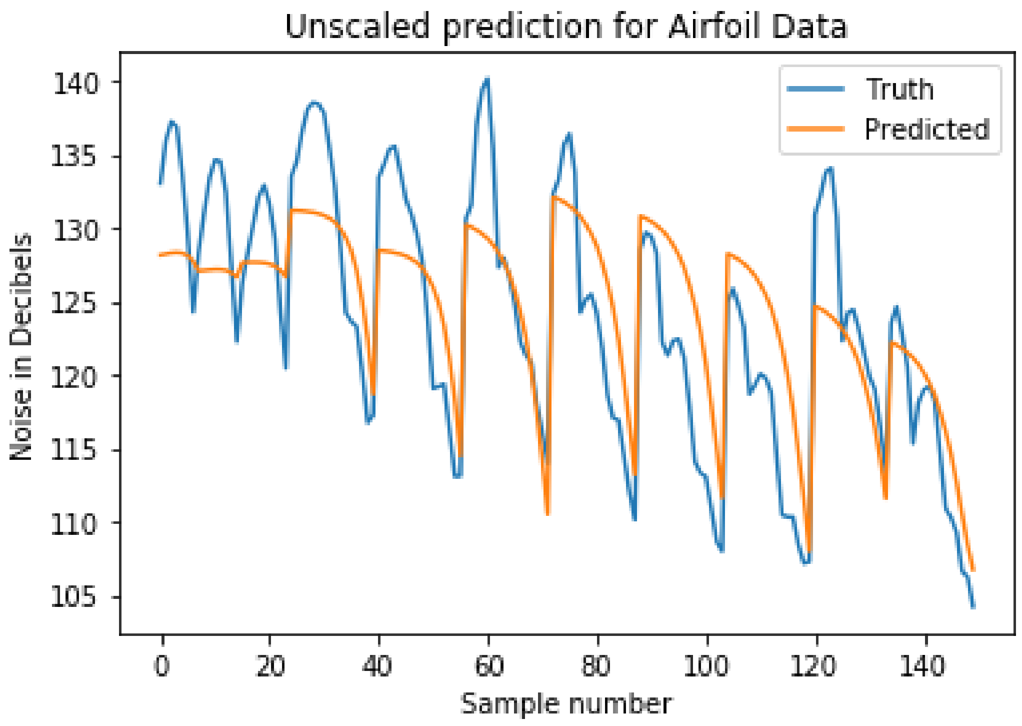
1.5.1 Definition : RBF

[19]:
```
def my_regressionRBF(trainX, testX, numOfOutputs):

    #error detection code
    if (trainX.shape[1] - testX.shape[1]) != numOfOutputs:
        print('Error - trainX({}), testX({}), nOutputs({}), \
              does NOT match the function requirement'.format(
            trainX.shape[1], testX.shape[1], numOfOutputs))

    #Create Folds - Split data into '5' folds
    nfolds = 5

    #number of samples provided for training
    ntrain = trainX.shape[0]
    indices = np.linspace(0, ntrain, nfolds + 1, dtype=int)

    nBasis = [20,30,50,80,120]
    sigmaSize = [.01,.05,.1,.5,1]
    errordata2 = np.zeros((len(indices) - 1, len(nBasis), len(sigmaSize)))

    xaverage = np.zeros((len(nBasis), len(sigmaSize)))

    for ii in range(len(indices) - 1):
        test_data = trainX[indices[ii]:indices[ii + 1]]
        train_data = trainX
        train_data = np.delete(train_data, np.s_[indices[ii]:indices[ii + 1]],0)

        #Taking feature matrix and output matrix from different folds
        train_data_ftr = train_data[:, :-numOfOutputs]
        train_data_out = train_data[:, -numOfOutputs:]

        #Taking Feature Matrix and output matrix from test data
        test_data_ftr = test_data[:, :-numOfOutputs]
        test_data_out = test_data[:, -numOfOutputs:]
        errordata = []

        for iii in range(len(nBasis)):
            for jjj in range(len(sigmaSize)):
                rbfFeature =␣
↪RBFSampler(gamma=sigmaSize[jjj],n_components=nBasis[iii],random_state=666)
                polyfit_ftr_matrix = rbfFeature.fit_transform(train_data_ftr)
                polyfit_testftr_matrix = rbfFeature.transform(test_data_ftr)
                #print(iii)
                #print(jjj)
                #print(ii)
                weight_matrix = linregression(polyfit_ftr_matrix,␣
↪train_data_out)
                test_predicted = np.matmul(polyfit_testftr_matrix,␣
↪weight_matrix)
                error = mseCalc(test_data_out, test_predicted)
                errordata2[ii, iii, jjj] = error


        df = pd.DataFrame(columns=["Fold", "Degree", "lambda", "error"],
                          data=errordata)

        if printDf == True:
            print(df)

    xaverage = np.mean(errordata2,axis=0)

    p = np.unravel_index(xaverage.argmin(), xaverage.shape)

    print("Minimum MSE is at nRBF = {} and gamma = {}".format(
        nBasis[p[0]], sigmaSize[p[1]]))

    #now let's train entire data with degree (degree[p[0]]) and lambda␣
↪(lamd[p[1]]) found in CV
    train_feature_matrix = trainX[:, :-numOfOutputs]
    train_output_matrix = trainX[:, -numOfOutputs:]
    rbfFeature =␣
↪RBFSampler(gamma=sigmaSize[p[0]],n_components=nBasis[p[1]],random_state=666)
    polyfit_ftr_matrix = rbfFeature.fit_transform(train_feature_matrix)
    polyfit_testftr_matrix = rbfFeature.fit_transform(testX)
    weight_matrix = linregression(polyfit_ftr_matrix, train_output_matrix)

    test_predicted = np.matmul(polyfit_testftr_matrix, weight_matrix)

    return test_predicted
```

1.5.2 Airfoil Self-Noise

[20]:
```
#load data
airfoil_data = np.loadtxt("airfoil_self_noise.dat")

#this variable needs to change for each database
numOfOutputs = 1

#normalize data
airfoil_data, featureMeans, featureStdevs = normalizeMatrix(airfoil_data)

#split into train and test data
train_data, test_data = my_train_test_split(airfoil_data, 10, False)

#call my_regression
printDf = False

#Doing it again with the RBF linear regression for giggles
test_data_prediction = my_regressionRBF(train_data, test_data[:, :
↪-numOfOutputs],numOfOutputs)

unScaledPrediction =␣
↪deNormalizeMatrix(test_data_prediction,featureMeans[-numOfOutputs:],␣
↪featureStdevs[-numOfOutputs:])
unScaledTarget = deNormalizeMatrix(test_data[:, -numOfOutputs:
↪],featureMeans[-numOfOutputs:], featureStdevs[-numOfOutputs:])

#print MSE of test_data
print('Unscaled MSE - {}'.format(mseCalc(unScaledPrediction, unScaledTarget)))

ax95 = plt.plot(unScaledTarget, label="Truth")
plt.plot(unScaledPrediction, label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("Noise in Decibels")
plt.title("Unscaled prediction for Airfoil Data")
plt.legend(loc = 'best')
plt.show()
```

```
Minimum MSE is at nRBF = 20 and gamma = 0.1
Unscaled MSE - [26.80914532]
```

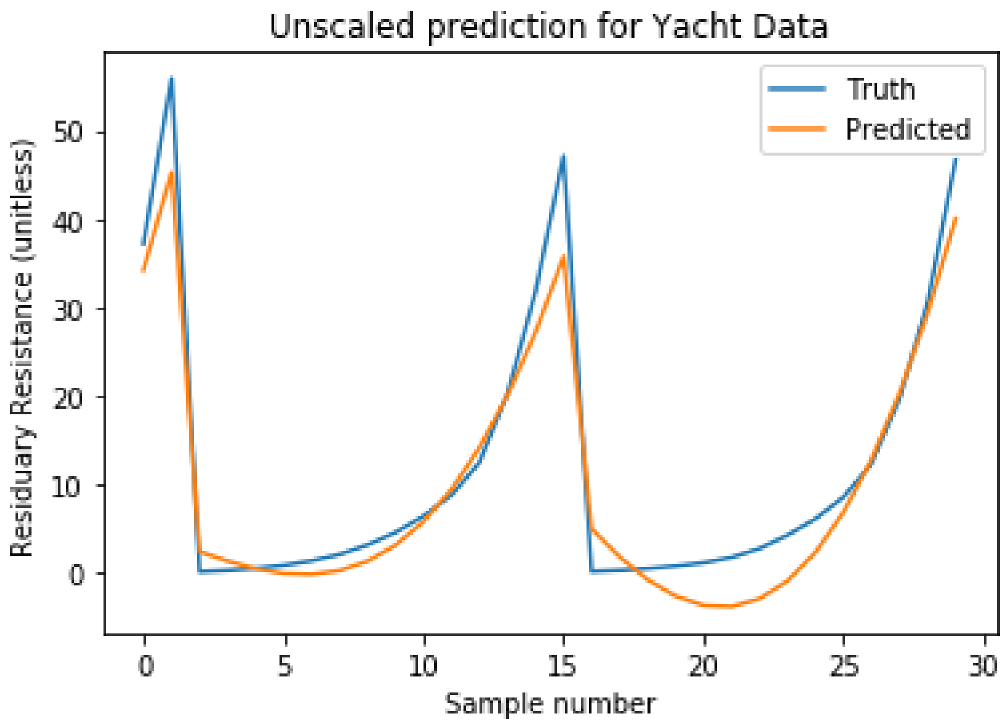


1.5.3 Yacht Hydrodynamics

[21]:
```
#load data
yacht_hydrodynamics_data = np.loadtxt("yacht_hydrodynamics.data")

#this variable needs to change for each database
numOfOutputs = 1

#normalize data
yacht_hydrodynamics_data, featureMeans, featureStdevs =␣
 ↪normalizeMatrix(yacht_hydrodynamics_data)

#split into train and test data
train_data, test_data = my_train_test_split(yacht_hydrodynamics_data, 10, False)

#call my_regression
printDf = False
```

```
#Doing it again with the RBF linear regression for giggles
test_data_prediction = my_regressionRBF(train_data, test_data[:, :
 ↪-numOfOutputs],numOfOutputs)

unScaledPrediction =␣
 ↪deNormalizeMatrix(test_data_prediction,featureMeans[-numOfOutputs:],␣
 ↪featureStdevs[-numOfOutputs:])
unScaledTarget = deNormalizeMatrix(test_data[:, -numOfOutputs:
 ↪],featureMeans[-numOfOutputs:], featureStdevs[-numOfOutputs:])

#print MSE of test_data
print('Unscaled MSE - {}'.format(mseCalc(unScaledPrediction, unScaledTarget)))

ax45 = plt.plot(unScaledTarget, label="Truth")
plt.plot(unScaledPrediction, label="Predicted")
plt.xlabel("Sample number")
plt.ylabel("Residuary Resistance (unitless)")
plt.title("Unscaled prediction for Yacht Data")
plt.legend(loc = 'best')
plt.show()
```

```
Minimum MSE is at nRBF = 20 and gamma = 0.05
Unscaled MSE - [17.05994176]
```


Unscaled prediction for Yacht Data
Truth
Predicted
Residuary Resistance (unitless)
Sample number