ELE435_Minh-Thi_Nguyen_HW5

October 21, 2019

1 ELE 435 Lab 5

1.1 Due Date: 10/21/2019 (Mon) 11:59 PM

1.1.1 Name: Minh-Thi Nguyen

[77]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
```

1.2 ** Regression

1.2.1 Q1) Housing Dataset:

In this part, we will be working with a dataset that contains house prices in suburbs of Boston. The data contains 14 columns, the first 13 of which correspond to different housing-related features , such as "CRIM" (crime rate per capita), "RM" (average number of rooms), "TAX" (tax rate), etc. A more comprehensive description can be found at https://archive.ics.uci.edu/ml/datasets/Housing. The last column corresponds to median value of owner-occupied homes in $1000's.

1-1) Using linear regression, find a vector "w" that specifies a weighting of each attribute for predicting the price of a house (median value in $1000's, similar to the last column of data). Print vector w.

[78]:
```
housing_data = np.loadtxt('housing.data')
X = np.matrix(housing_data[:,0:13])
y = np.matrix(housing_data[:,13]).T
```

[79]:
```
print(X.shape)
print(y.shape)
```

```
(506, 13)
(506, 1)
```

[80]:
```
F = X
#minimize Fw-y
# linear regression w = (F^TF)^(-1)F^Ty
```

```
w = np.matmul(np.matmul(np.linalg.inv(np.matmul(F.T,F)),F.T),y)
```

1-2) Using vector w from the previous part, print the mean squared error of predicting house prices.

[81]:
```
n = y.size

#error vector
E = (y - np.matmul(X,w))
#magnitude of error vector (error^2)
sum_error = np.matmul(E.T,E)

#mean squared error
mse = 1/n*sum_error

print(mse)
```

```
[[24.16609933]]
```

1.2.2 Q2) Physicochemical Properties of Protein Tertiary Structure Data Set:

In this part, we will be working with a new dataset that quantifies protein tertiary structure. There are 45730 decoys with size varying from 0 to 21 angstrom. Each example contains 9 features. More information can be found at https://archive.ics.uci.edu/ml/datasets/Physicochemical+Properties+of+Protein+Tertiary+Structure#.

First, we will import data into numpy using the following code. Afterwads, data is split into trainingg/test sets. The goal is to learn a vector "w" (using linear regression on the training data) to predict the size of a test-decoy based on its 9 features.

[82]:
```
Protein_data = np.loadtxt('Protein.txt')
train_X = np.matrix(Protein_data[0:40000,1:]) # Dictionary that contains␣
 ↪features for training samples
train_y = np.matrix(Protein_data[0:40000,0]).T # Size of decoy for each training␣
 ↪sample (a real number)

test_X = np.matrix(Protein_data[40000:,1:]) # Dictionary that contains features␣
 ↪for test samples
test_y = np.matrix(Protein_data[40000:,0]).T # Size of decoy for each test␣
 ↪sample (a real number)
```

Q2-1) Based on train_X and train_y, find the corresponding vector "w" (using linear regression) and print it.

[87]:
```
#function
def linear_regression(F,y):
    return(np.matmul(np.matmul(np.linalg.inv(np.matmul(F.T,F)),F.T),y))
```

[88]:
```
w_train = linear_regression(train_X,train_y)
print(w_train)
```

```
[[ 1.82919845e-03]
 [ 7.93316713e-04]
 [ 2.46155609e+01]
 [-1.05302864e-01]
 [-3.85363213e-06]
 [-2.50966502e-02]
 [-1.15479761e-04]
 [ 1.46576918e-02]
 [-2.61397924e-02]]
```

Q2-2) Using vector "w" from the previous part, predict the size of decoys for test samples (test_X). What is the mean squared error between the predicted values and the actual sizes (test_y)?

[89]:
```
#predict the size of the decoys for test samples
y_test = np.matmul(test_X,w_train)
```

[90]:
```
#mean squared error

def mean_squared_error(y1,y2):
    n = y1.size
    E = (y1 - y2)
    sum_error = np.matmul(E.T,E)
    mse = 1/n*sum_error
    return(mse)
```

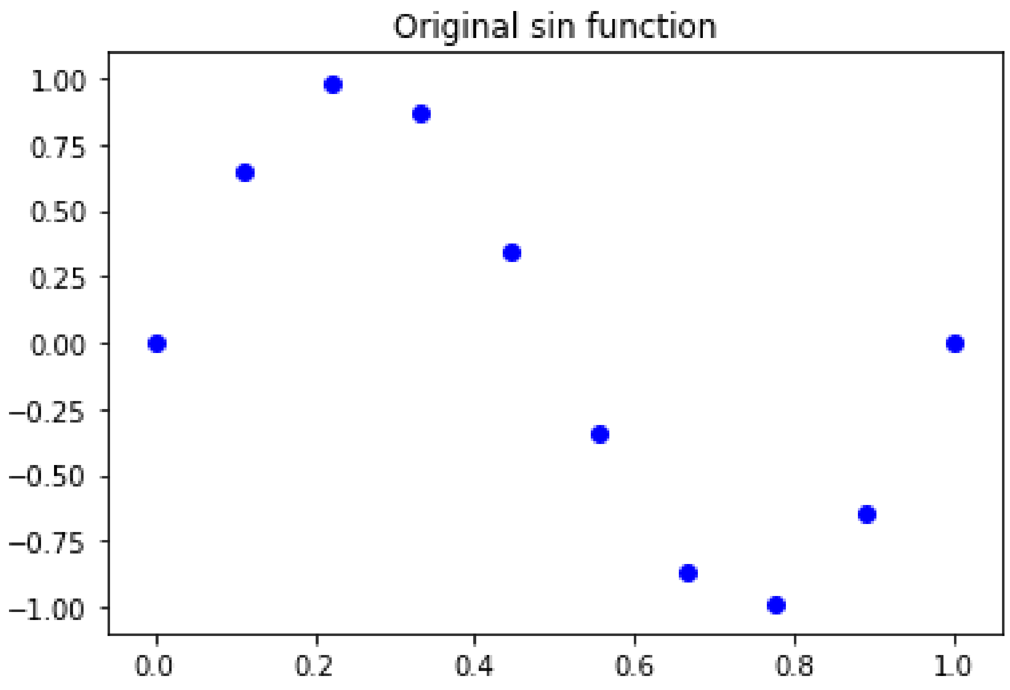
[91]:
```
mse_protein = mean_squared_error(test_y,y_test)
print(mse_protein)
```

```
[[27.01412467]]
```

1.2.3 Q3) Understanding Overfitting:

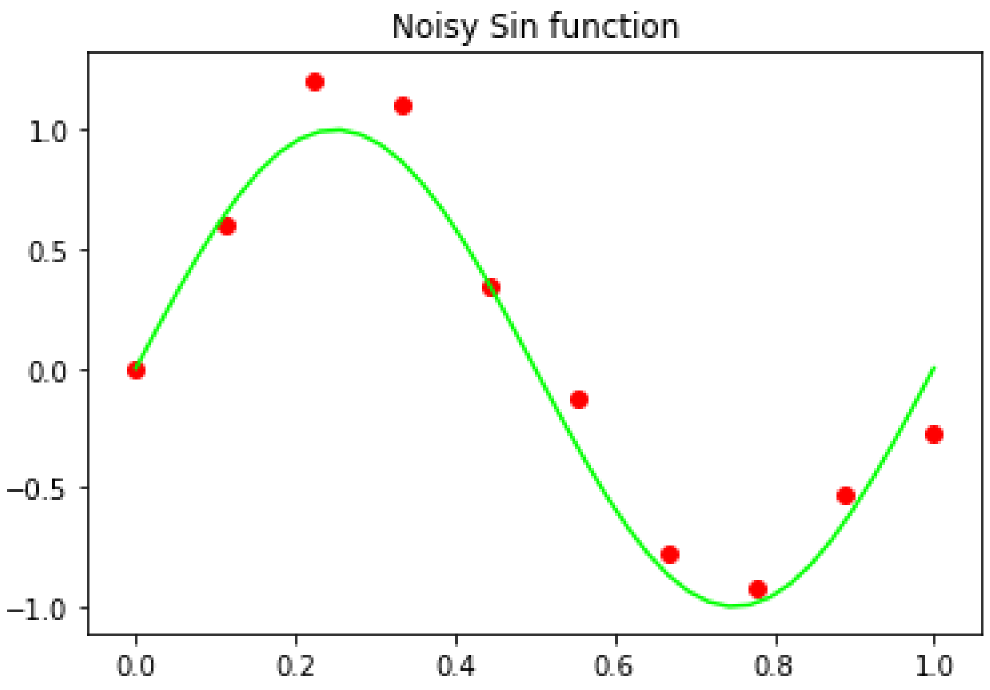
In this part, we will work on an example that helps us understand the problems with overfitting. First, we will pick 10 samples of a sin wave using the following commands.

[92]:
```
Num_points = 10
x = np.linspace(0, 1, Num_points)
signal = np.sin(2*np.pi*x)
plt.figure()
plt.scatter(x, signal, color = (0,0,1))
plt.title('Original sin function')
plt.show()
```



Then, we are going to add Gaussian noise to the original samples.

[93]:
```
sigma = 0.15
mu = 0.1
noise = sigma * np.random.randn(Num_points) + mu
corrupt_signal = signal + noise

plt.figure()
plt.scatter(x, corrupt_signal, color = (1,0,0))
plt.plot(np.linspace(0, 1, Num_points*4),np.sin(2*np.pi*(np.linspace(0, 1,␣
 ↪Num_points*4)))
         ,color=(0,1,0))
plt.title('Noisy Sin function')
plt.show()
```

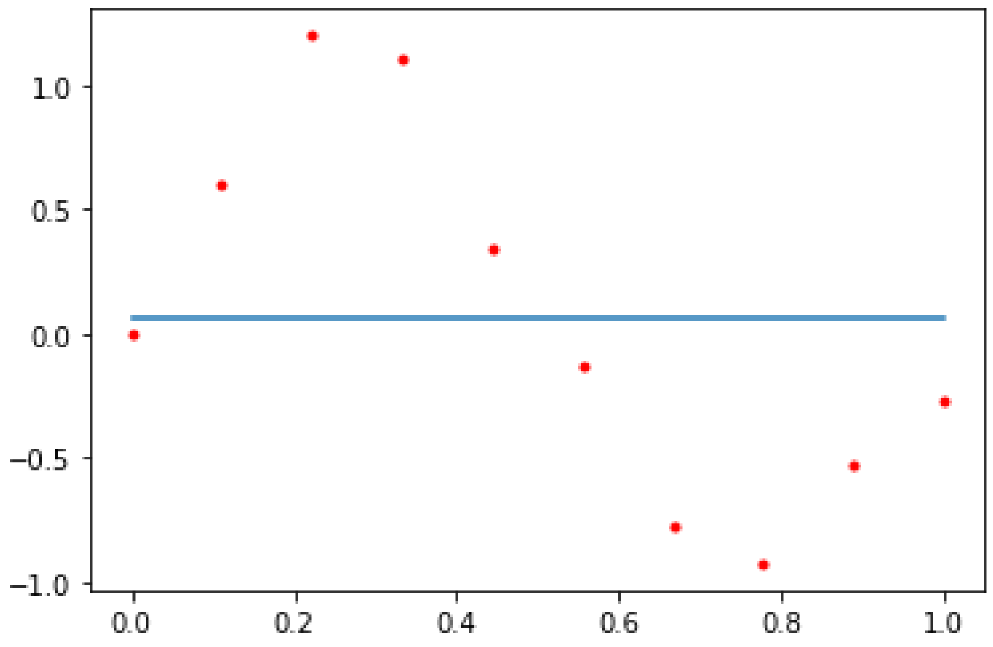


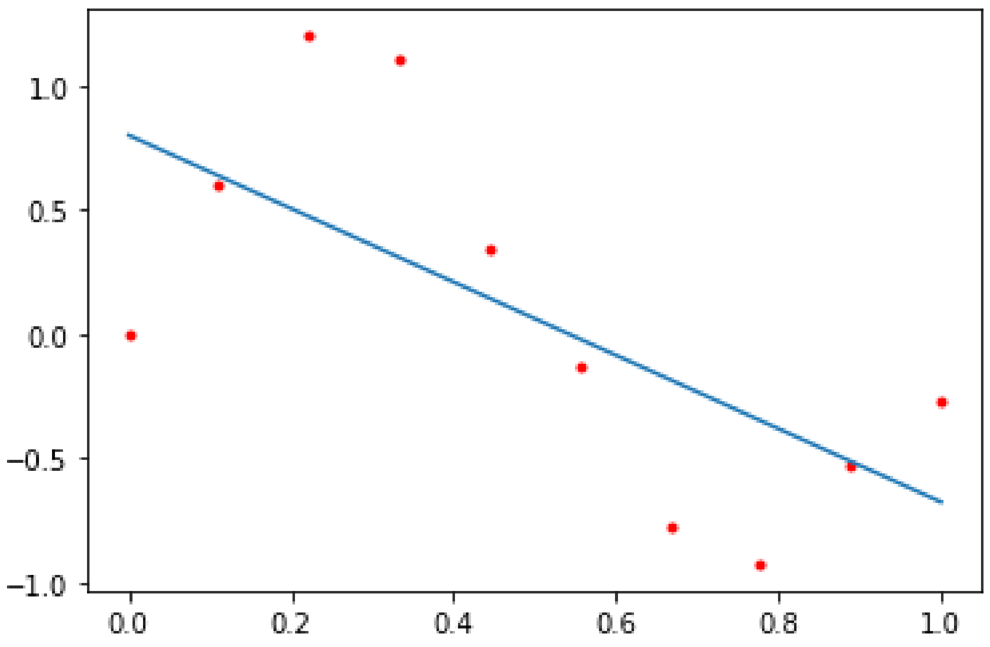
Now, using regression, fit polynomials of different degrees (degree=0 up to degree=9) to the noisy samples (noisy samples come from "corrupt_signal" in the code above). Plot the corresponding polynomials along with the original sin function (10 different plots, in total). Describe what happens as you fit a more complex polynomial to your data.

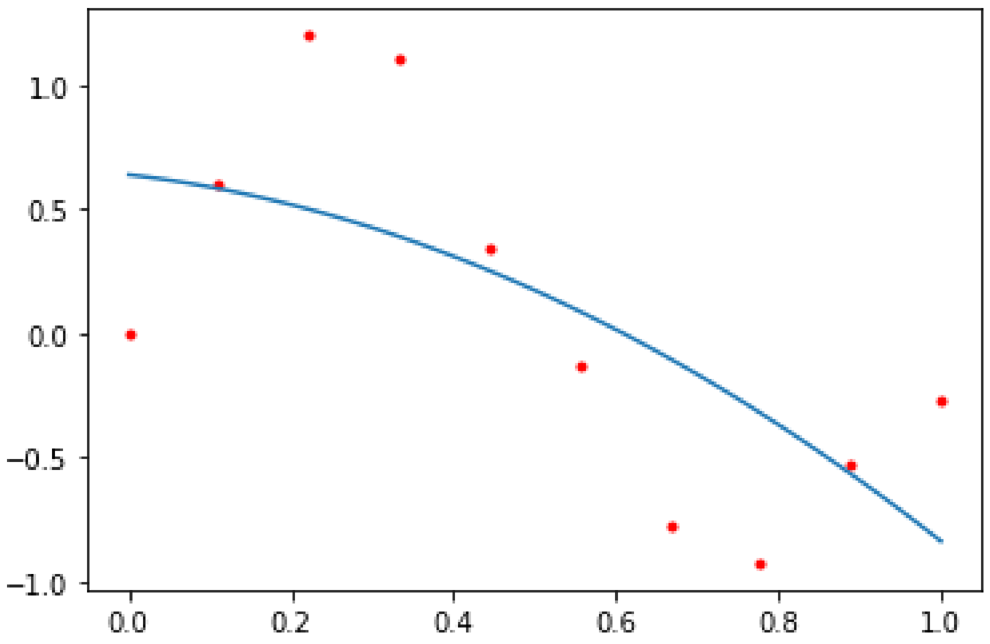
[94]:
```
def create_matrix(v,k):
    n = v.size
    matrix = np.zeros((n,k))
    for i in range(k):
        matrix[:,i] = v**i
    return(matrix)
```

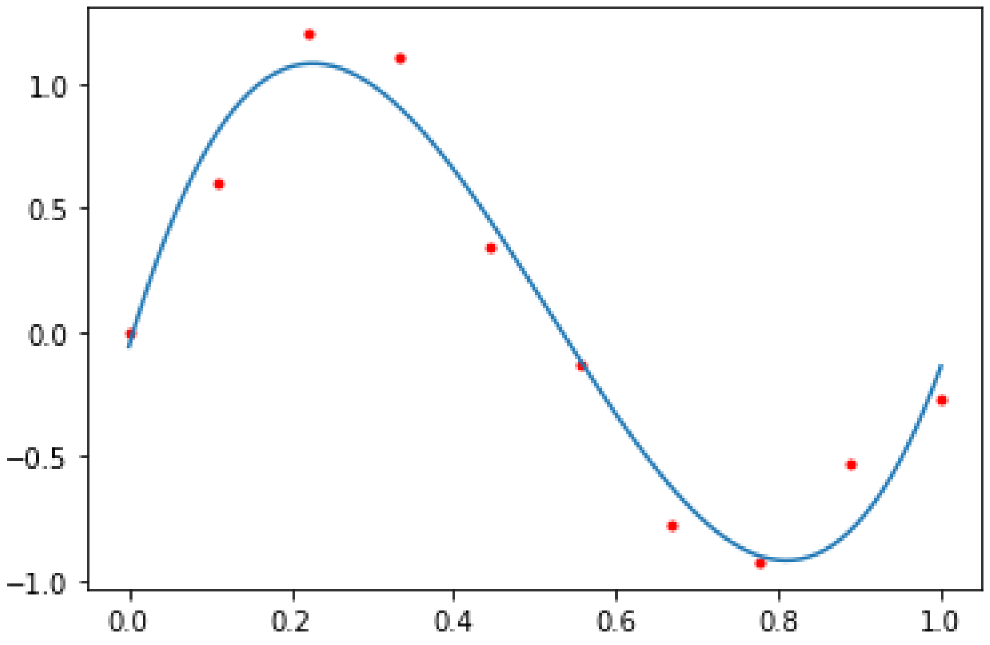
[95]:
```
import numpy as np
from matplotlib import pyplot as plt

def polynomial_define(x, coefficients):
    n = len(coefficients)
    y = 0
    for i in range(n):
        y += coefficients[i]*x**i
    return y

#x = np.linspace(0, 9, 10)
#coeffs = [1, 2, 3, 4, 5]
#plt.plot(x, PolyCoefficients(x, coeffs))
#plt.show()
```

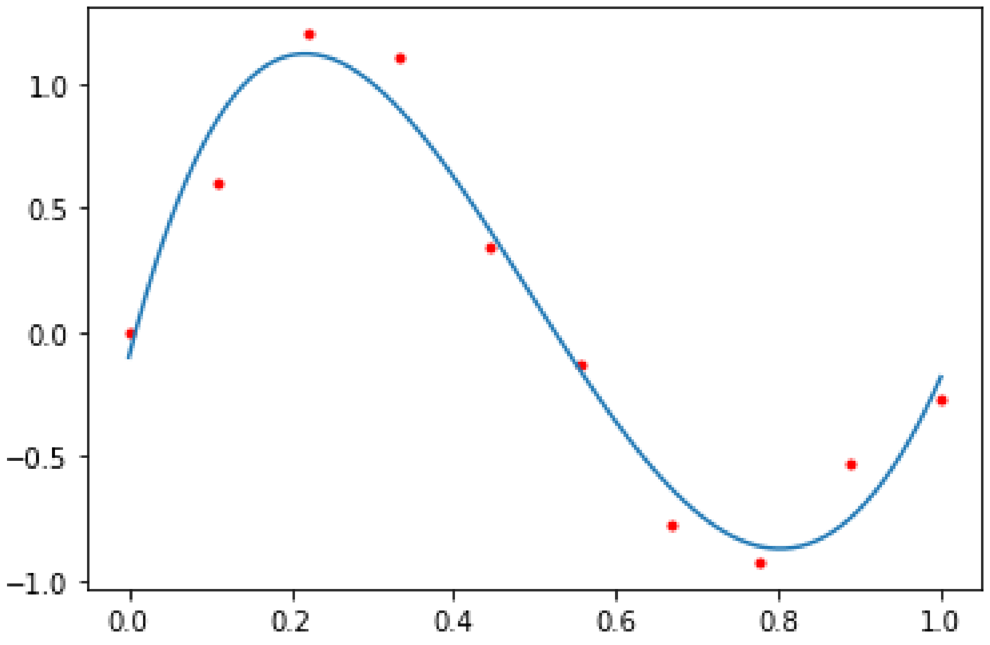
[96]:
```
degree = np.arange(10)

for i in degree:
    X = create_matrix(x,i+1)
    w_sin = linear_regression(X,corrupt_signal)
    x_vector = np.linspace(0, 1, 100)
    poly = polynomial_define(x_vector,w_sin.T)

    plt.plot(x,corrupt_signal, 'r.')
    plt.plot(x_vector,poly)
    plt.show()
```

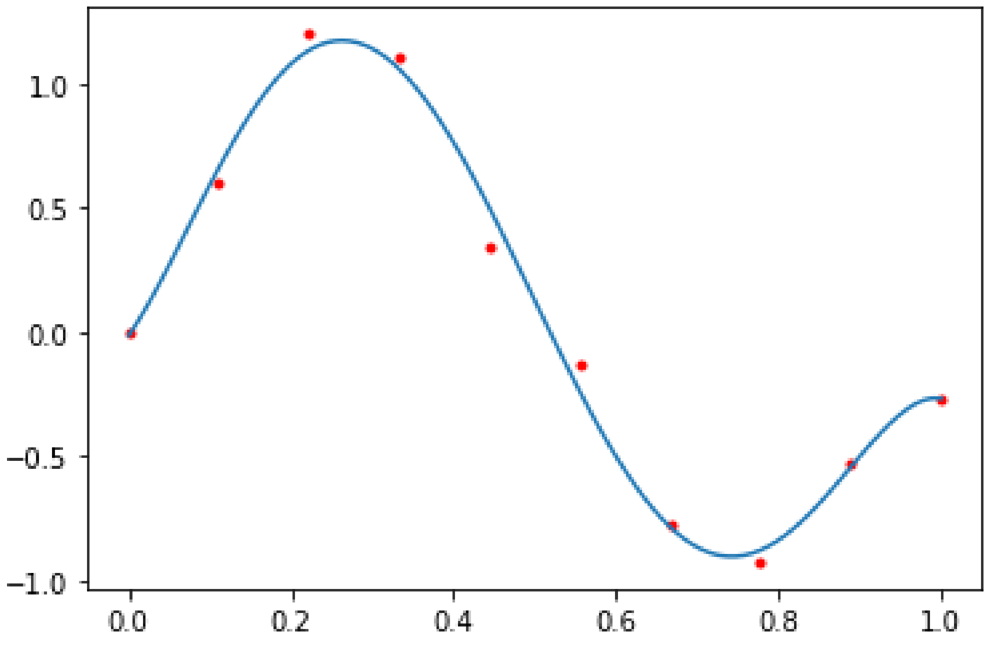


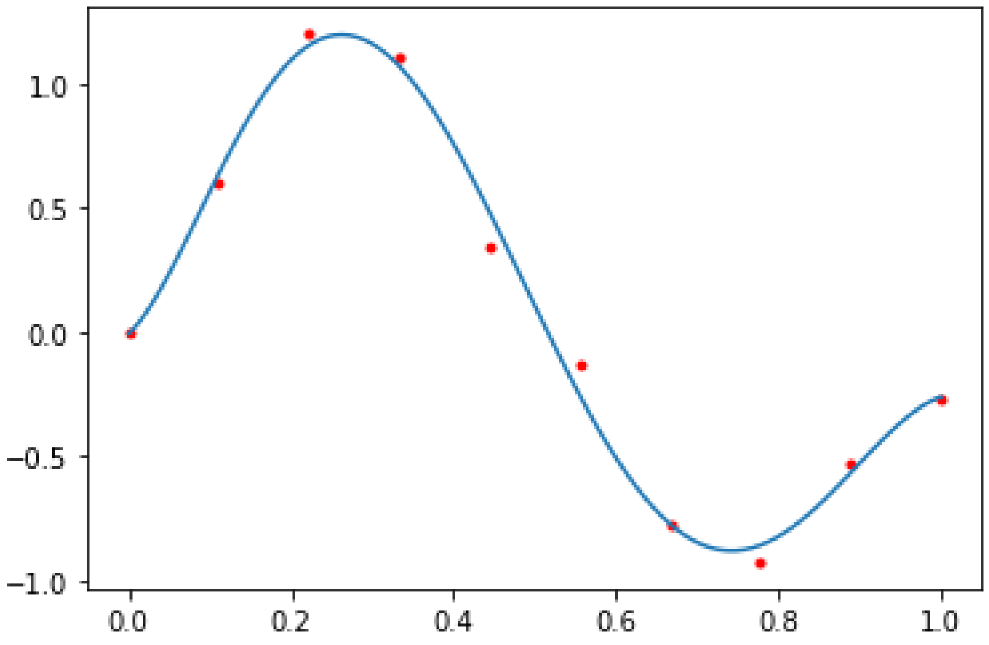

1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

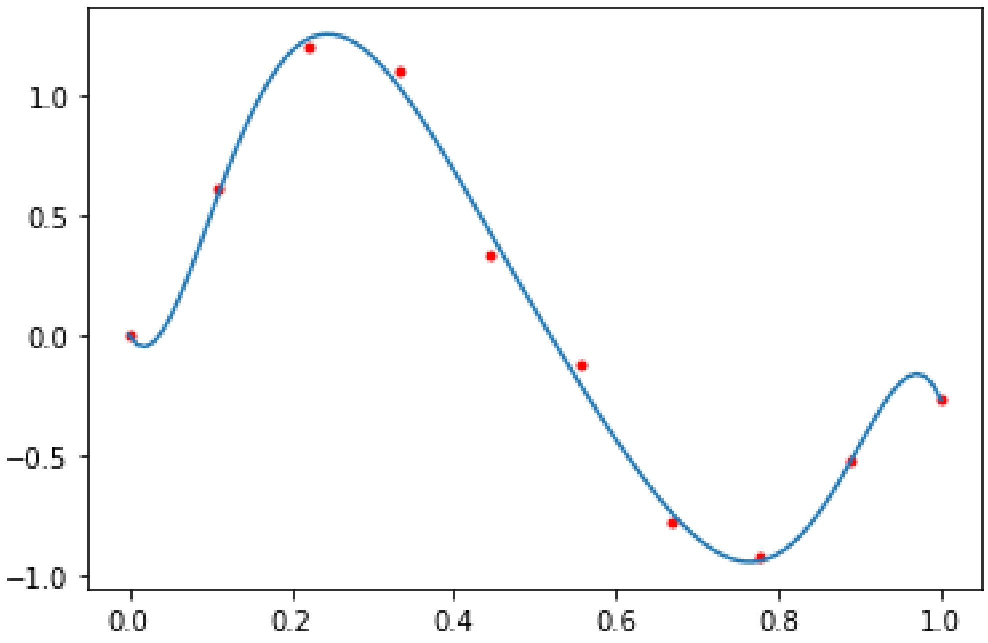

1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

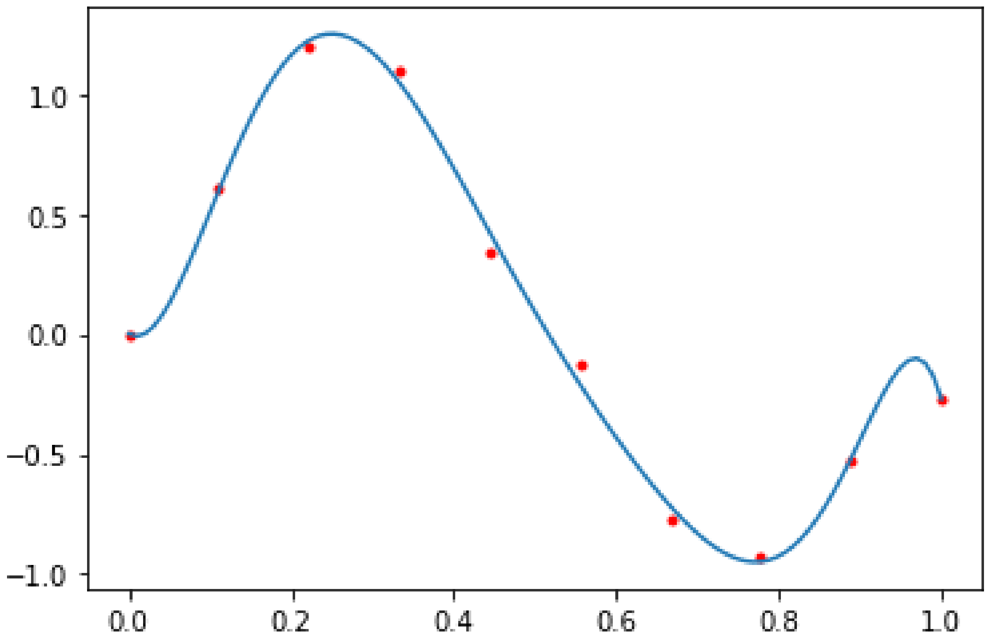

1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0

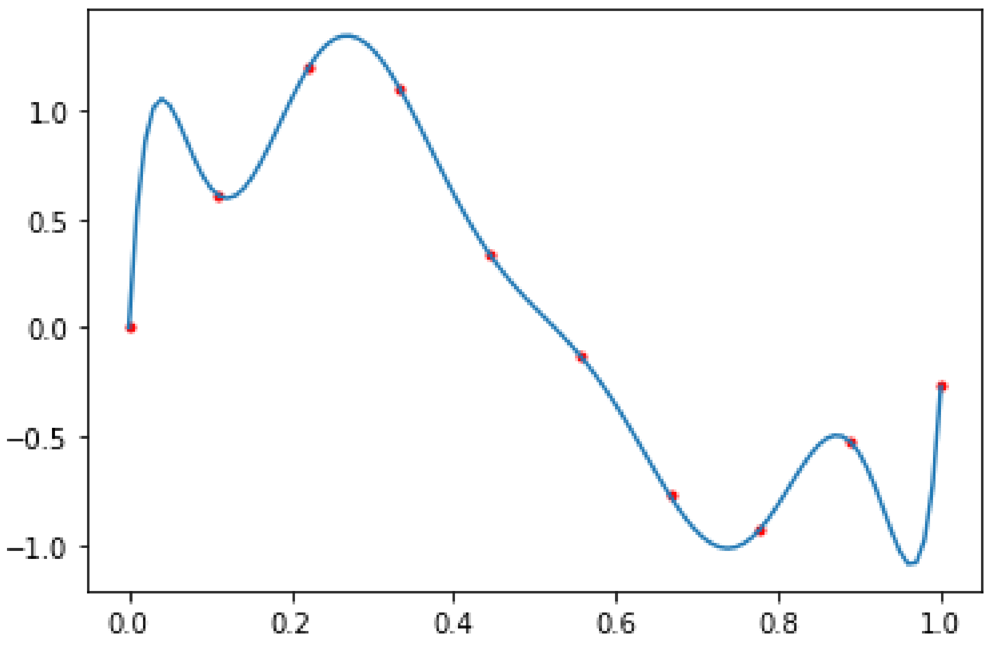

1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0


1.0
0.5
0.0
−0.5
−1.0
0.0
0.2
0.4
0.6
0.8
1.0



1.2.4 Q4) Collaborative Representation Based Classification:

The goal of this part is to use regression for classifiying data. We will be working with the subset of MNIST that was used for HW1. The training data contains 10,000 samples of different digits. Let's call it matrix D (of dimension 784 * 10000). The first 1000 columns of D correspond to digit 0 (D_0), the next 1000 correspond to digit 1 (D_1), etc.

Now, given any new example (x), we would like to represent it as a linear combination of columns of D (hence, the name representation based classification). This can be acheived by finding a vector w (of dimension 10,000) that satisfies; $w = \arg\min \; \|Dw - x\|_2$. The first 1000 elements of w (w_0) quantify how much of each column from digit 0 are needed to represent x. Similarly, the next 1000 elements (w_1) correspond to weights on D_1, etc. (Hint: Getting w directly from normal equation of D is time-consuming, SVD will help to speed up the calculation)

Next, prediction of pixel values of any test image (x) based only on examples of a particual digit i can be found using $y'_i = D_i \times w_i$. Then, k-th digit that yields the lowest mean squared prediction error (i.e., $k = \arg\min \; \|y - y'_i\|_2$) will determine the label of x.

Follow the procedure above to predict the labels of each test example. What is the corresponding testing accuracy?

[97]:
```
# Regression-based classification
train_data = np.load('MNISTcwtrain1000.npy')
train_data = train_data.astype(dtype='float64')
test_data = np.load('MNISTcwtest100.npy')
test_data = test_data.astype(dtype='float64')

train_data = train_data/255.0
test_data = test_data/255.0
```

[98]:
```python
u, s, vh = np.linalg.svd(train_data, full_matrices=False)
W_matrix = np.zeros((10000,1000))

for i in range(test_data.shape[1]):
    x = test_data[:,i] #test vector x
    w = np.matmul(vh.T,np.matmul(np.diag(1/s),np.matmul(u.T,x)))
    W_matrix[:,i] = w
```

[99]:
```python
accuracy = 0;

for j in range(test_data.shape[1]):
    # loop through each test vector
    y = test_data[:,j]
    w_j = W_matrix[:,j]

    mse_array = np.zeros(10)

    #compare for each D_i
    for i in range(10):
        D_i = train_data[:,i*1000:(i+1)*1000]
        w_i = w_j[i*1000:(i+1)*1000]

        y_i = np.matmul(D_i,w_i)

        mse_array[i] = mean_squared_error(y_i,y)

    index_min = np.where(mse_array == np.amin(mse_array))

    if index_min == np.floor(j/100):
            accuracy = accuracy + 1

print(accuracy/1000)
```

```
0.86
```