hw8_1

April 17, 2023

[7]:
```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
```

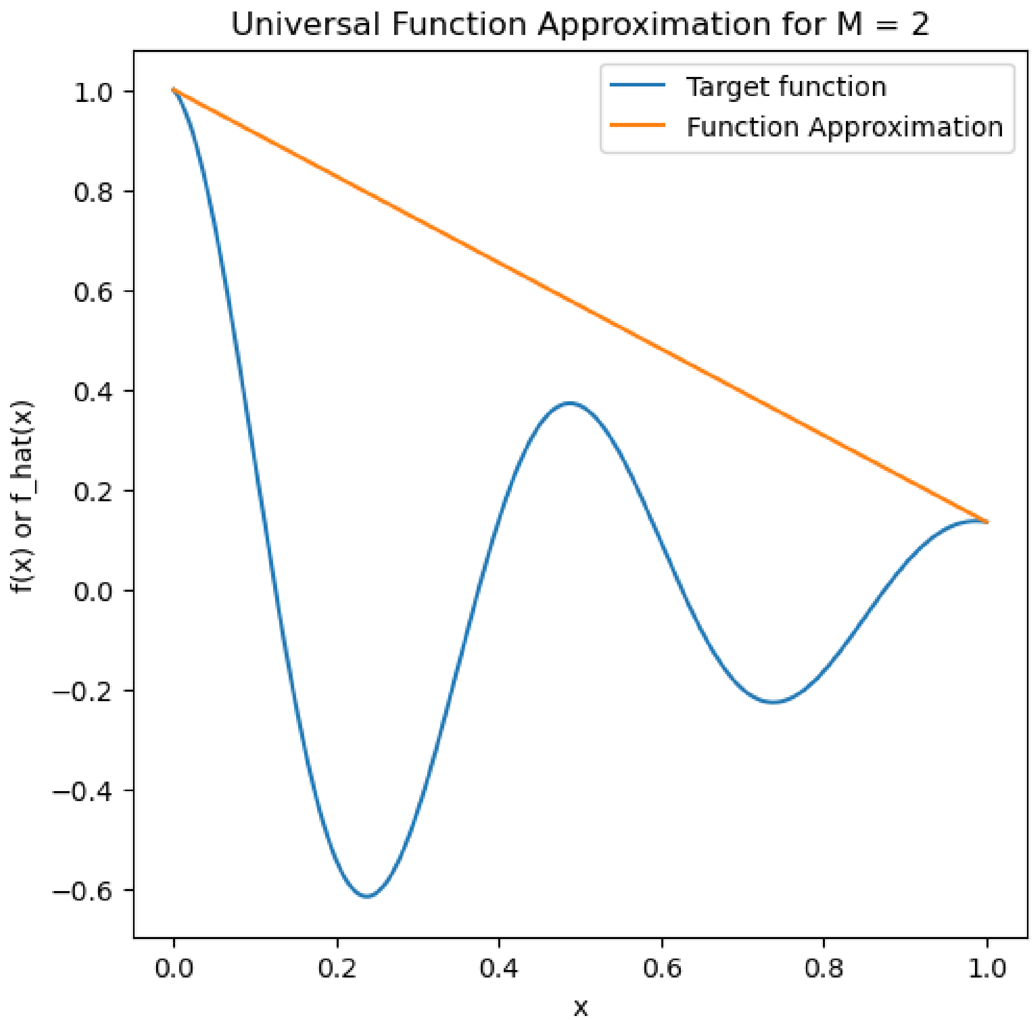
[16]:
```
def grid(M):
    x = np.zeros(M)
    for i in range(M):
        x[i] = (i / (M - 1) )
    return x
# print(grid(2))
# print(grid(4))
```

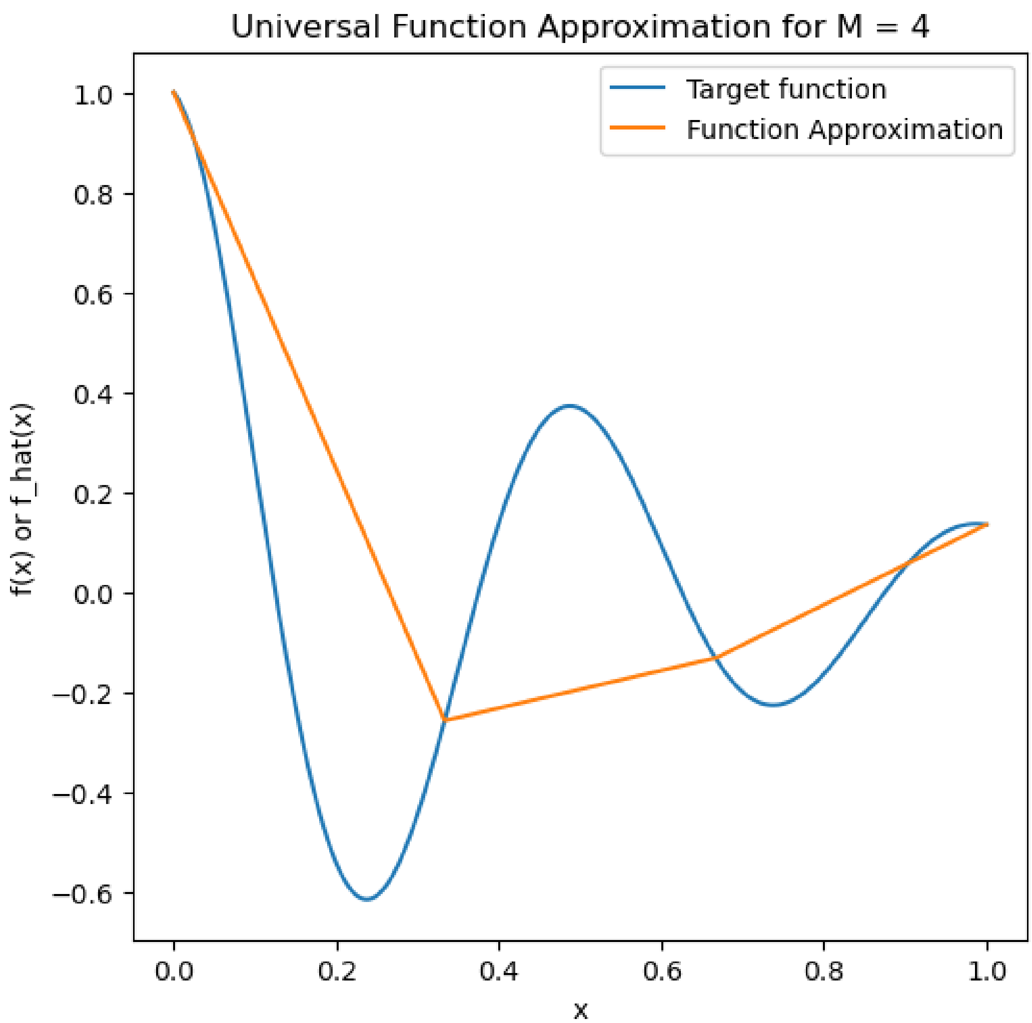
[5]:
```
def original_function(x):
    return np.exp(-2*x) * np.cos(4*np.pi*x)
```

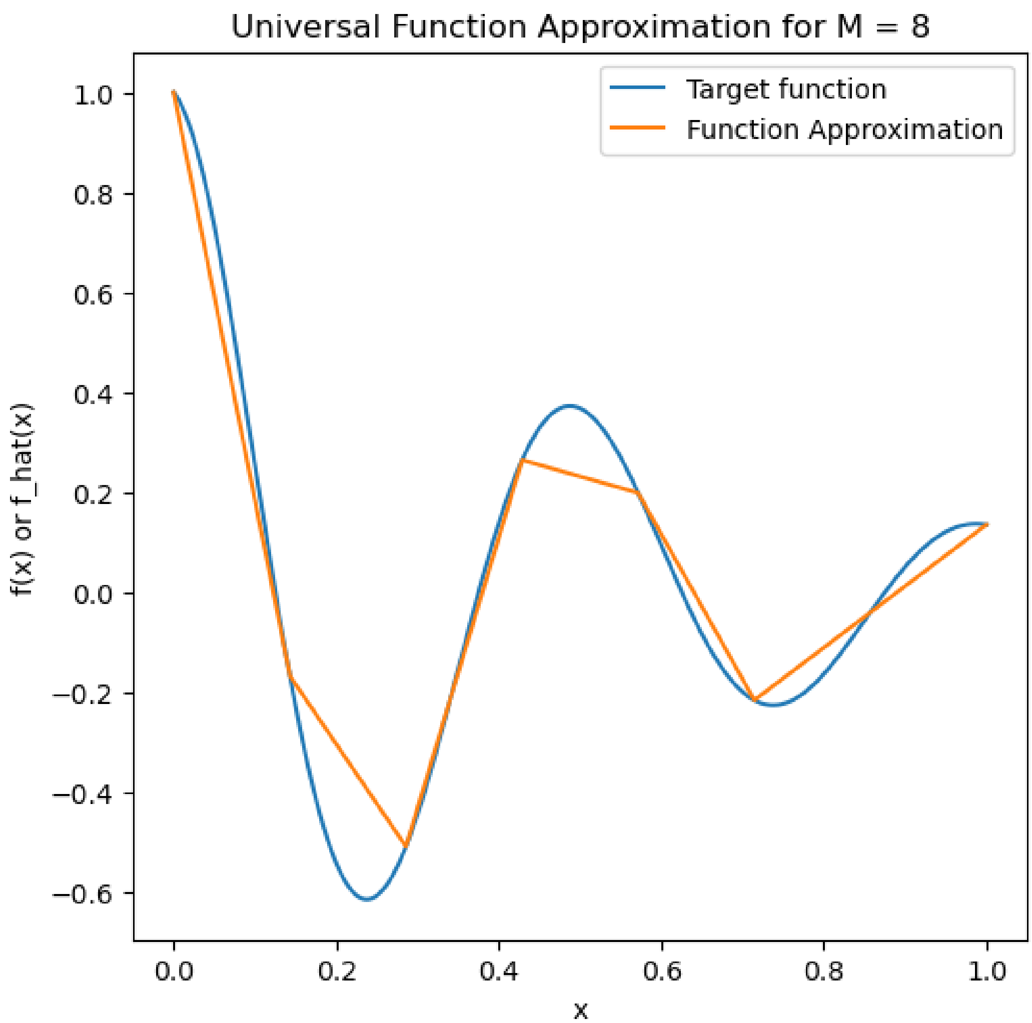
[21]:
```
def vx(x, w_m0):
    return np.maximum(0, x + w_m0)
```

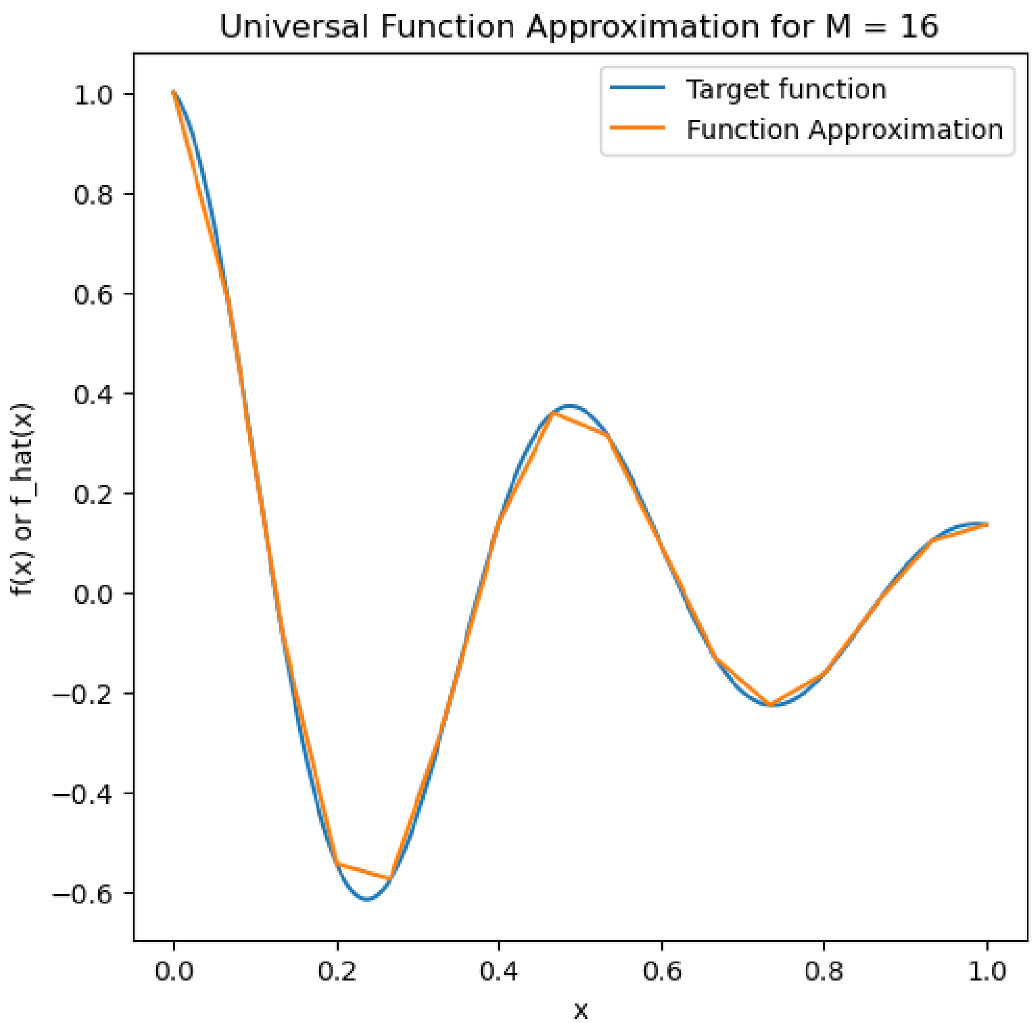
[48]:
```
def approximation_function():
    # Range of M
    M_range = [2, 4, 8, 16]

    for M in M_range:
        # define grid for approximated x, y
        x_grid = grid(M)
        y_grid = original_function(x_grid)

        # interpolate grid on target function
        x_range = np.linspace(0, 1, 2000)
        f_hat = np.interp(x_range, x_grid, y_grid)

        # plot approximation and target function on same graph for each M
        fig = plt.figure(figsize=(6, 6))
        plt.plot(x_range, original_function(x_range), label = 'Target function')
        plt.plot(x_range, f_hat, label = 'Function Approximation')
        plt.xlabel('x')
```

```
        plt.ylabel('f(x) or f_hat(x)')
        plt.title(f'Universal Function Approximation for M = {M}')
        plt.legend()
        plt.show()

approximation_function()
```




Universal Function Approximation for M = 4
Target function
Function Approximation
f(x) or f_hat(x)
x
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8
1.0

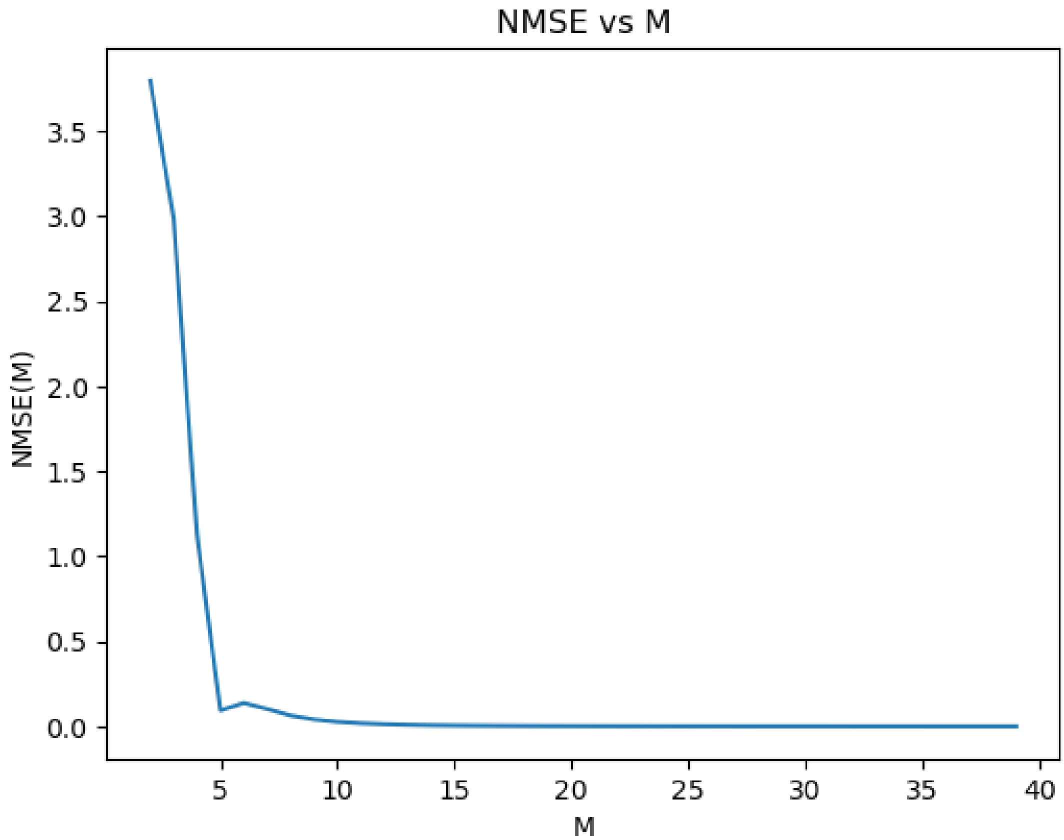

Universal Function Approximation for M = 8
Target function
Function Approximation
f(x) or f_hat(x)
x
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.0
0.2
0.4
0.6
0.8
1.0

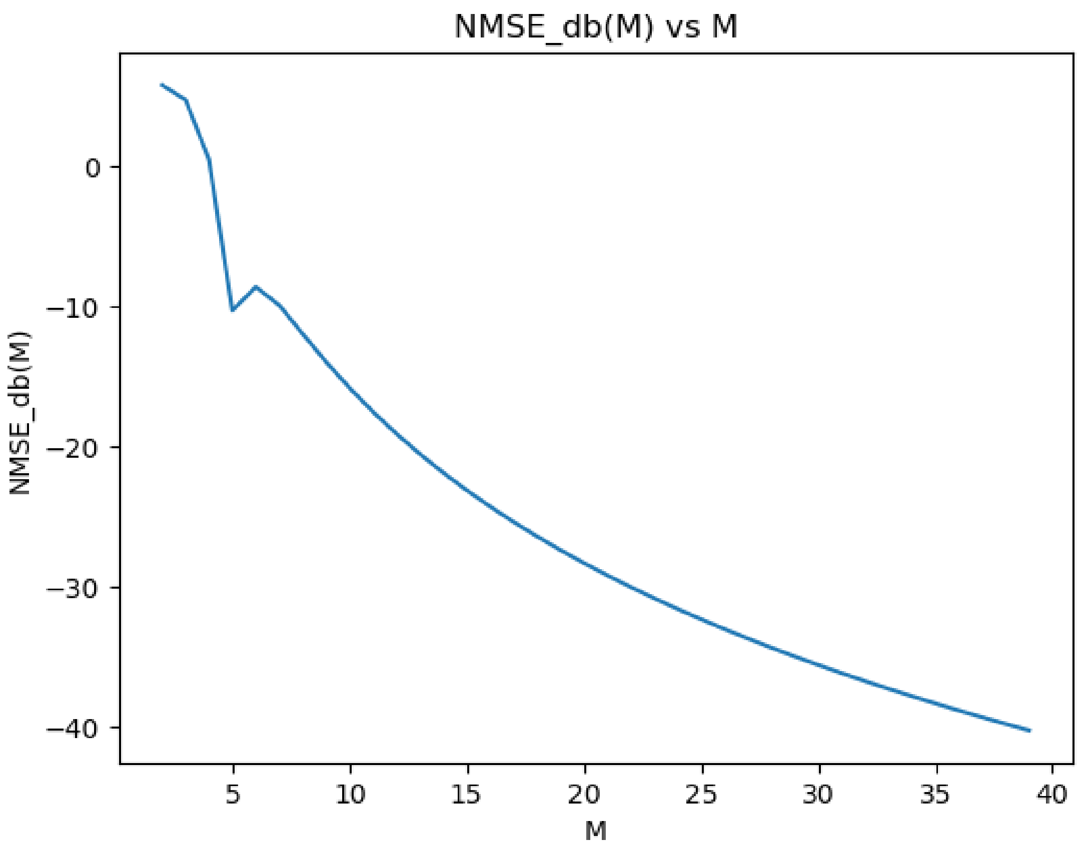


[36]:
```
## this function is from Prof. Chugg's nmse_01 notebook
## https://github.com/keithchugg/ee559_spring2023/blob/main/hw_helpers/nmse_01.
↪py

def normalized_mse_01(f, f_hat, x_grid, G=10000):
    # f: target function
    # f_hat: values of f_hat on the grid x_grid on [0,1]
    # x_grid a "coarse" grid on [0,1].  This has M point from the approximation.
    # G: grid size for a fine grid used to approximate the integral.

    x_fine =  np.linspace(0, 1, G)                     # create the fine grid
    f_fine = f(x_fine)                                 # evaluate f on the fine␣
↪grid
    f_hat_fine = np.interp(x_fine, x_grid, f_hat)    # interpolate f_hat to the␣
↪fine grid
```

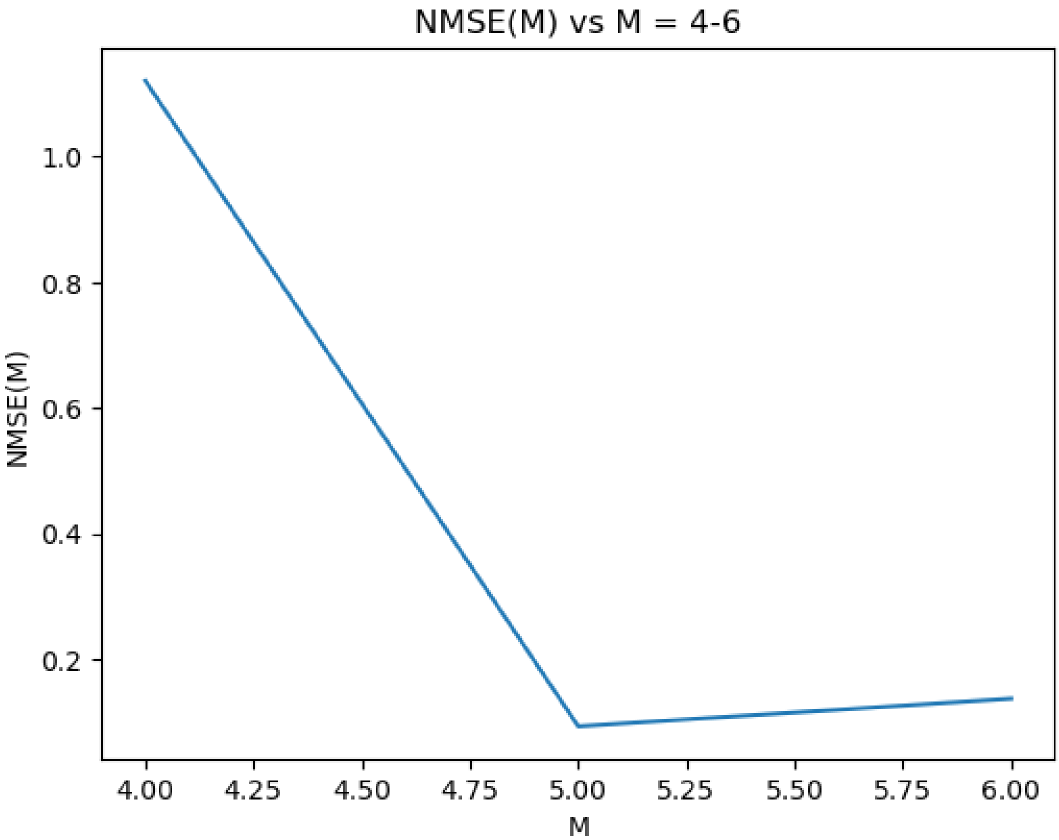
```
    sq_error = (f_fine - f_hat_fine) ** 2           # compute squared error
    mse = np.mean(sq_error)                          # this is a scalar multiple␣
↪of the integral (approximately)
    ref = np.mean(f_fine ** 2)                       # Energy in target; off by␣
↪same scalar as mse
    return mse / ref                                 # scalar values cancel
```

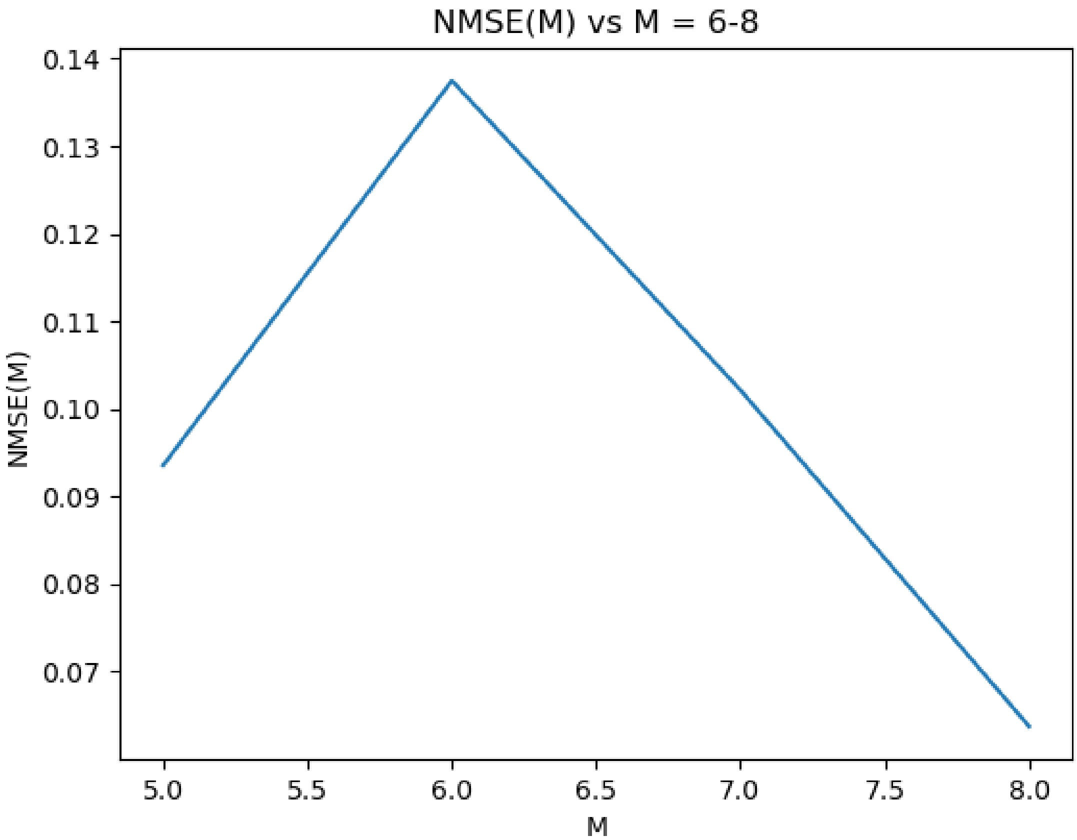
[59]:
```
def plot_NMSE():
    # Range of M

    res = []
    res_db = []
    M = 2
    M_range = []
    while(M >= 2):
        M_range.append(M)

        # define grid for approximated x, y
        x_grid = grid(M)
        f_hat = original_function(x_grid)

        nmse = normalized_mse_01(original_function, f_hat, x_grid)
        res.append(nmse)
        res_db.append(10 * np.log10(nmse))
        if(10 * np.log10(nmse) < -40):
            break;
        M += 1
    plt.title("NMSE vs M")
    plt.xlabel("M")
    plt.ylabel("NMSE(M)")
    plt.plot(M_range,res)
    plt.show()

    plt.title("NMSE_db(M) vs M")
    plt.xlabel("M")
    plt.ylabel("NMSE_db(M)")
    plt.plot(M_range,res_db)
    plt.show()

    return res, M_range

NMSE, M_range = plot_NMSE()
```


NMSE vs M
NMSE(M)
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
5
10
15
20
25
30
35
40
M



[72]:
```
plt.title("NMSE(M) vs M = 4-6")
plt.xlabel("M")
plt.ylabel("NMSE(M)")
plt.plot(M_range[2:5], NMSE[2:5])
plt.show()
plt.title("NMSE(M) vs M = 6-8")
plt.xlabel("M")
plt.ylabel("NMSE(M)")
plt.plot(M_range[3:7], NMSE[3:7])
plt.show()
print(M_range[-1])
```


NMSE(M) vs M = 4-6
NMSE(M)
1.0
0.8
0.6
0.4
0.2
4.00
4.25
4.50
4.75
5.00
5.25
5.50
5.75
6.00
M

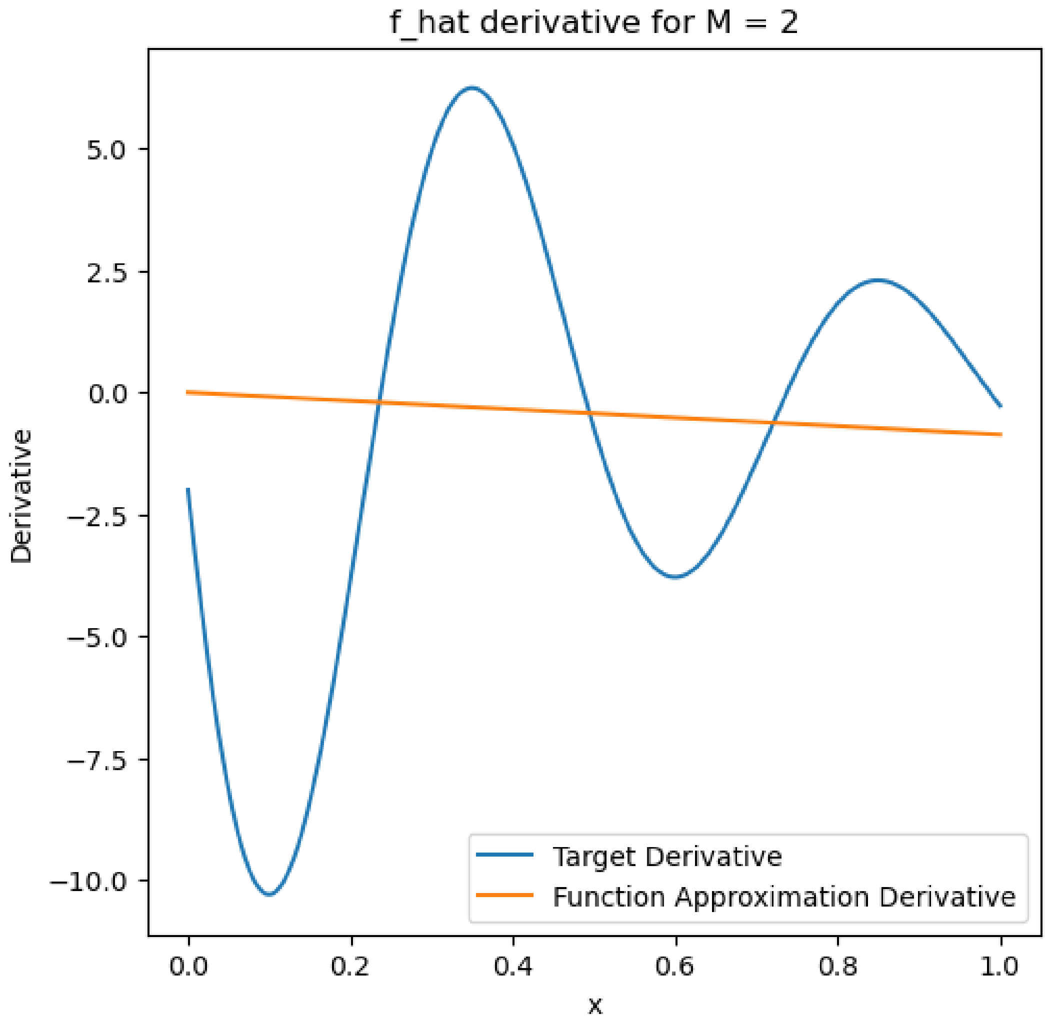


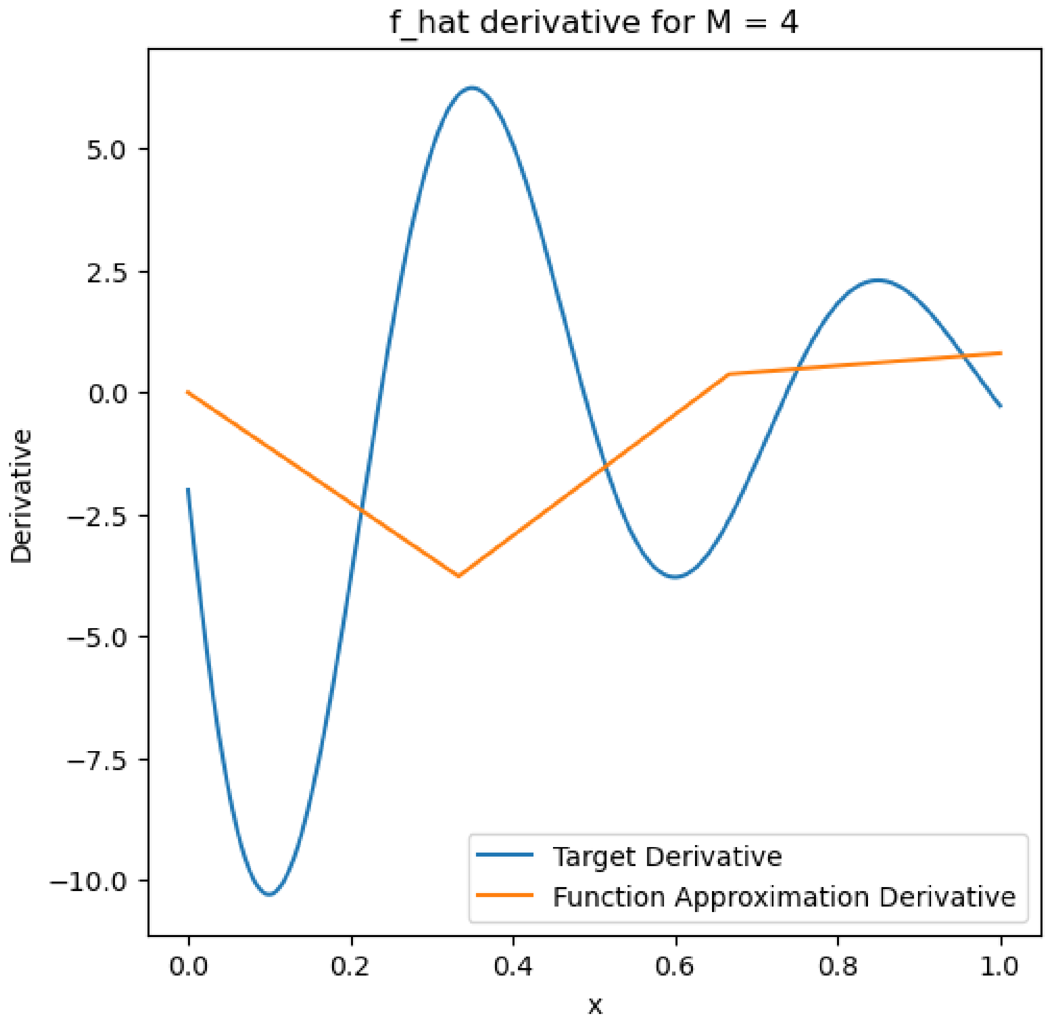
39

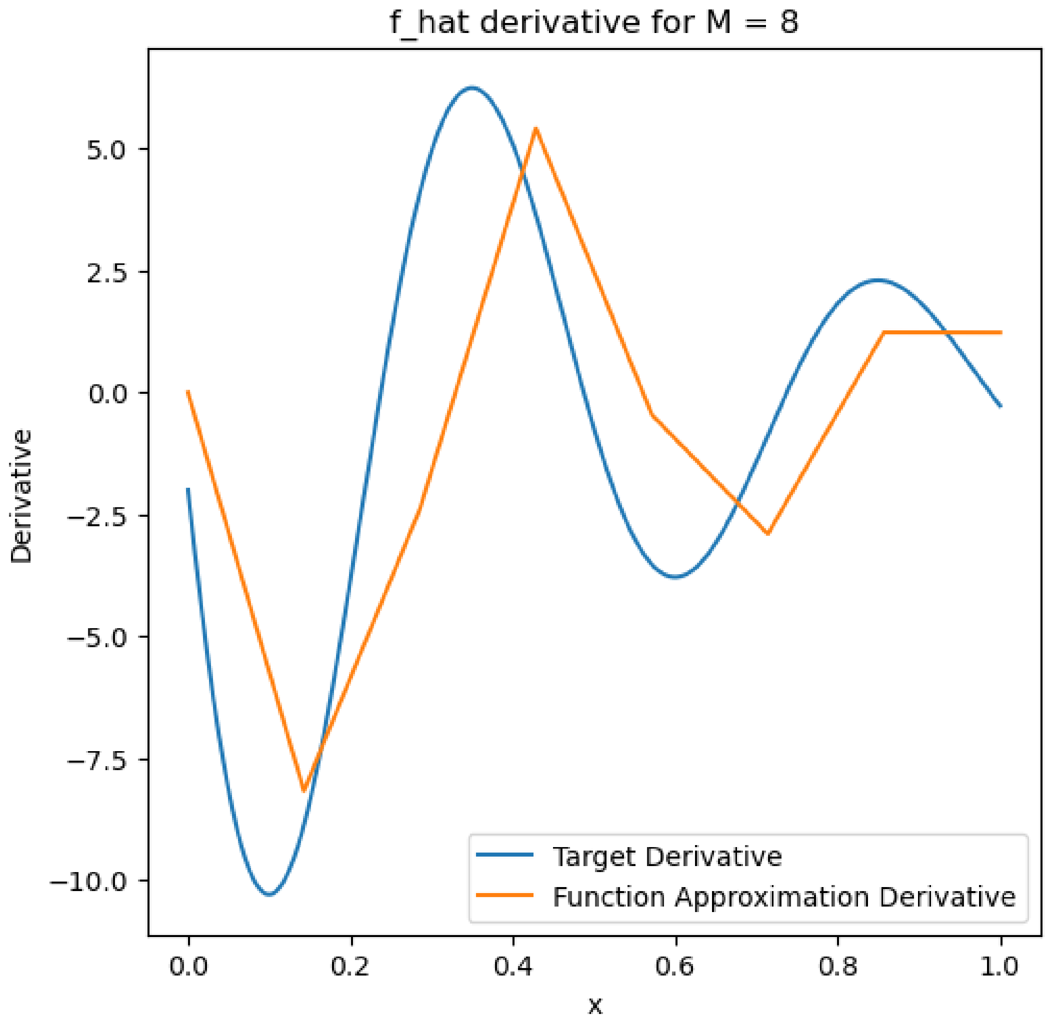
[97]:
```
def derivative(x, h=1e-6):
    return (original_function(x + h) - original_function(x)) / h
```

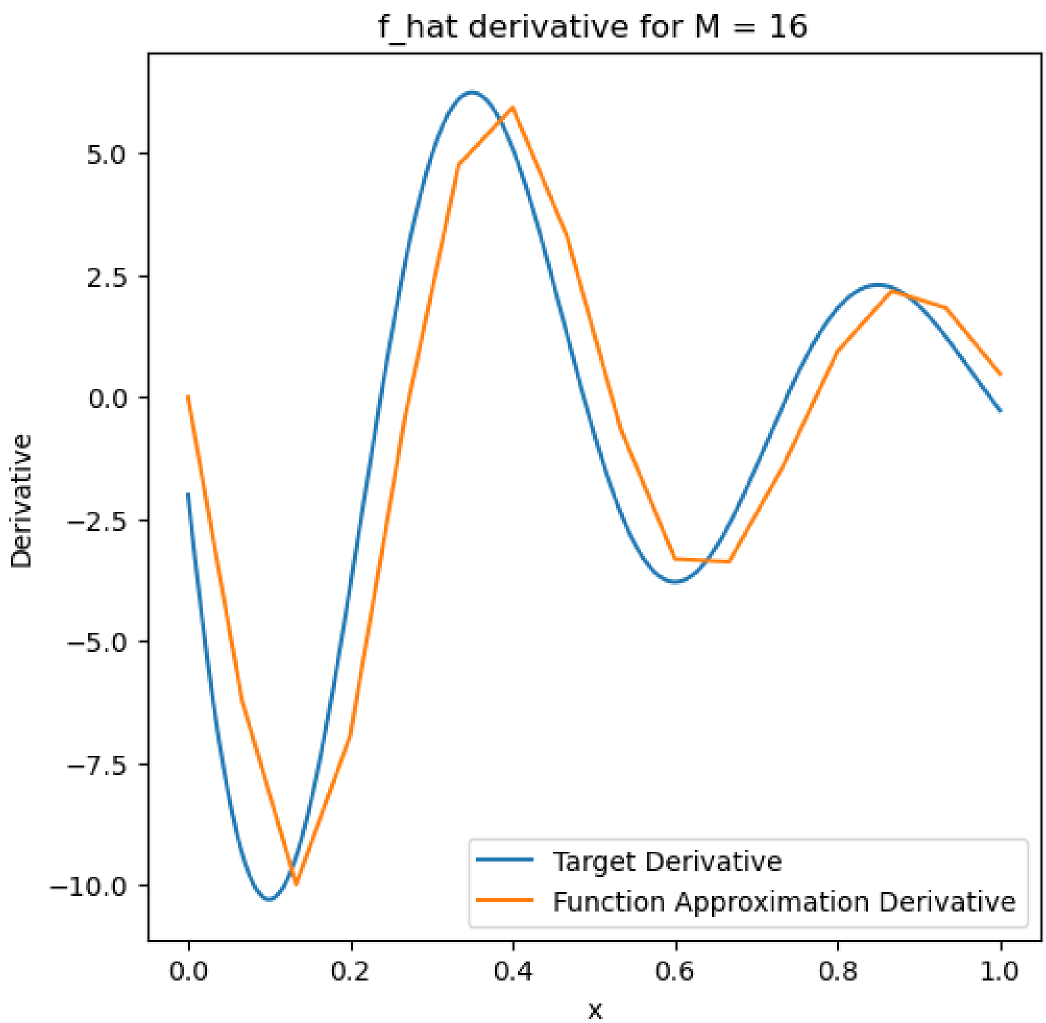
[98]:
```
def approximation_derivative_function():
    # Range of M
    M_range = [2, 4, 8, 16]

    for M in M_range:
        # define grid for approximated x, y
        x_grid = grid(M)
        y_grid = original_function(x_grid)
        f_hat_der = [0]
        for i in range(len(x_grid)):
            if(i == 0):
                continue
            f_hat_der.append((y_grid[i] - y_grid[i - 1]) / (x_grid[i] -␣
↪x_grid[i - 1]))
```

```
        # Compute derivative on target function
        x_range = np.linspace(0, 1, 2000)
        y_der = derivative(x_range)

#         f_hat = np.interp(x_range, x_grid, y_grid)

        # plot approximation and target function on same graph for each M
        fig = plt.figure(figsize=(6, 6))
        plt.plot(x_range,y_der,label = 'Target Derivative')
        plt.plot(x_grid,f_hat_der, label = 'Function Approximation Derivative')
        plt.xlabel('x')
        plt.ylabel('Derivative')
        plt.title(f'f_hat derivative for M = {M}')
        plt.legend()
        plt.show()

approximation_derivative_function()
```


f_hat derivative for M = 2
Derivative
x
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0.0
0.2
0.4
0.6
0.8
1.0
Target Derivative
Function Approximation Derivative

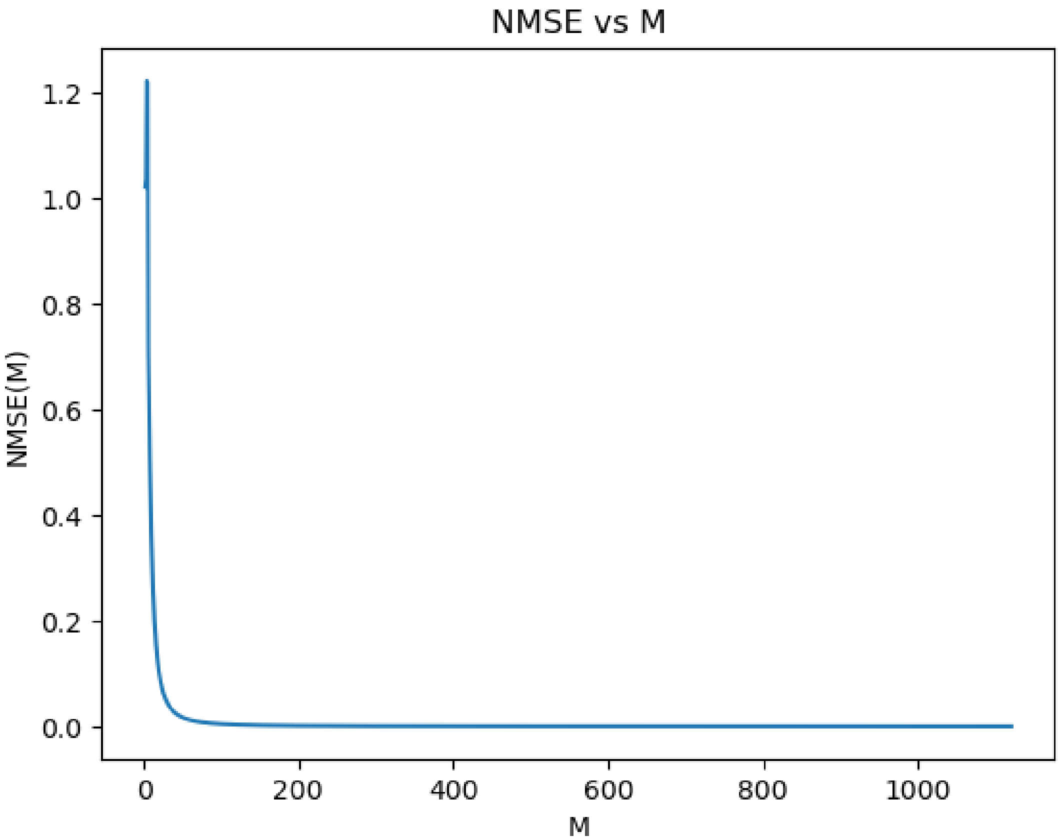

f_hat derivative for M = 4
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
Derivative
0.0
0.2
0.4
0.6
0.8
1.0
x
Target Derivative
Function Approximation Derivative

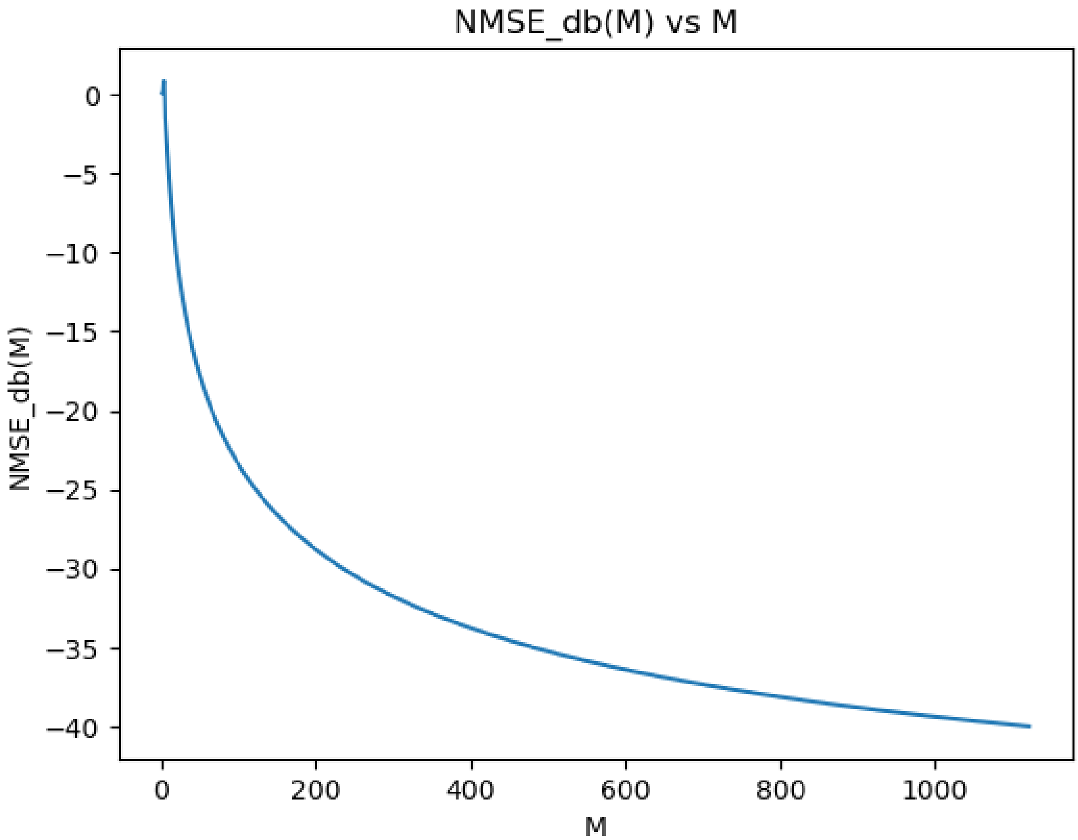

f_hat derivative for M = 8
Derivative
x
Target Derivative
Function Approximation Derivative
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
0.0
0.2
0.4
0.6
0.8
1.0



[100]:
```
def plot_deri_NMSE():
    # Range of M

    res = []
    res_db = []
    M = 2
    M_range = []
    while(M >= 2):
        M_range.append(M)

        # define grid for approximated x, y
        x_grid = grid(M)
        y_grid = original_function(x_grid)

        f_hat_der = [0]
```

```
        for i in range(len(x_grid)):
            if(i == 0):
                continue
            f_hat_der.append((y_grid[i] - y_grid[i - 1]) / (x_grid[i] -␣
 ↪x_grid[i - 1]))


        # Compute derivative on target function
        x_range = np.linspace(0, 1, 2000)
#         y_der = derivative(original_function,x_range)

        nmse = normalized_mse_01(derivative, f_hat_der, x_grid)
        res.append(nmse)
        res_db.append(10 * np.log10(nmse))
        if(10 * np.log10(nmse) < -40):
            break;
        M += 1
    plt.title("NMSE vs M")
    plt.xlabel("M")
    plt.ylabel("NMSE(M)")
    plt.plot(M_range,res)
    plt.show()

    plt.title("NMSE_db(M) vs M")
    plt.xlabel("M")
    plt.ylabel("NMSE_db(M)")
    plt.plot(M_range,res_db)
    plt.show()

    return res, M_range

NMSE, M_range = plot_deri_NMSE()
```


NMSE vs M
NMSE(M)
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
M


NMSE_db(M) vs M
NMSE_db(M)
M
0
−5
−10
−15
−20
−25
−30
−35
−40
0
200
400
600
800
1000

```
[ ]:
```

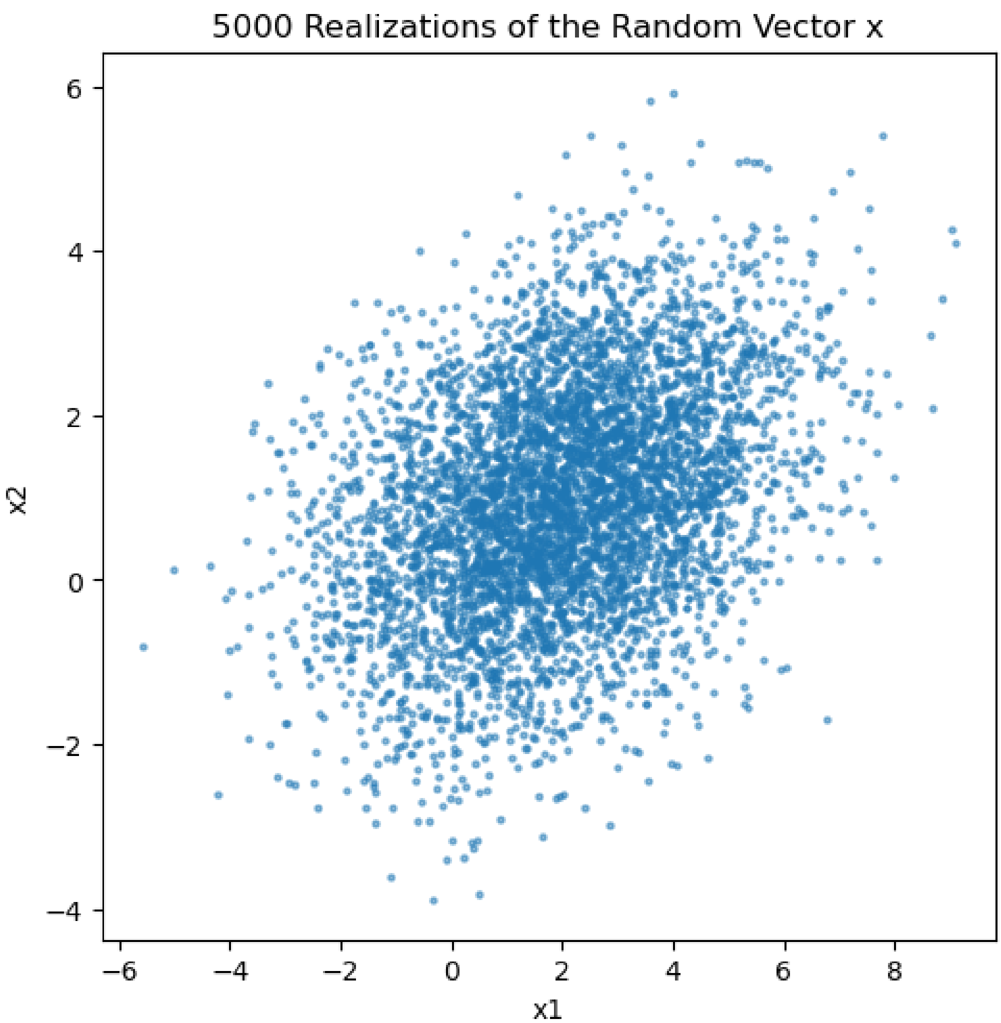
hw8_2

April 17, 2023

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
def plot_scatter(mean, cov, size):

    # Generate the random realizations
    realizations = np.random.normal(0,1,size=(size, 2))
    eigvalues, eigvectors = np.linalg.eig(cov)

    # Transform the realizations
    transformed = np.dot(eigvectors, np.sqrt(np.diag(eigvalues))) @␣
 ↪realizations.T

    # Add the mean to each realization
    realizations = transformed.T + mean

    # Scatter plot
    plt.figure(figsize=(6,6))
    plt.scatter(realizations[:, 0], realizations[:, 1], s=5, alpha=0.5)
    plt.title('5000 Realizations of the Random Vector x')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.show()
    return realizations
```

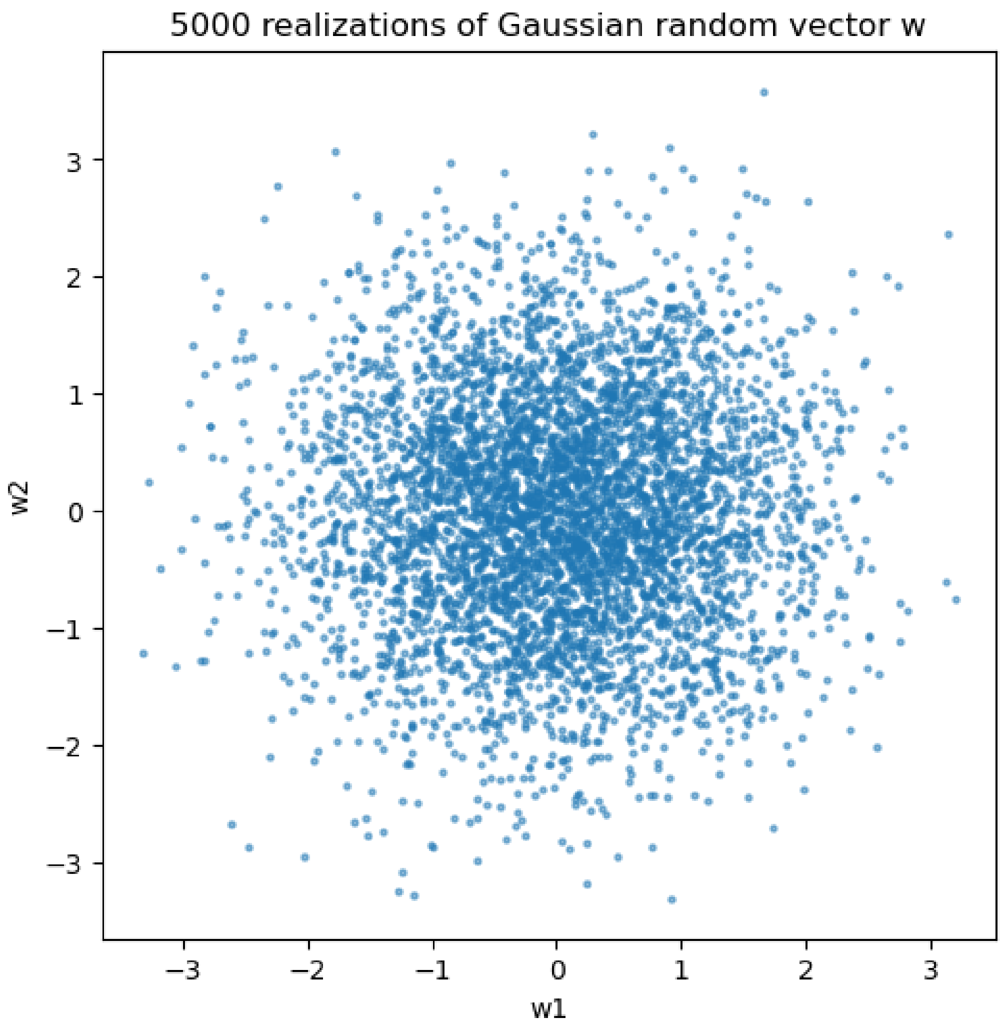
[3]:
```
mean_x = [2, 1]
cov_x = [[4, 1], [1, 2]]
x = plot_scatter(mean_x, cov_x, 5000)
```




```
[4]: def white_Gaussian_vector(mean, cov, size):

         # Generate the random realizations
         realizations = np.random.normal(0,1,size=(size, 2))
     #     eigvalues, eigvectors = np.linalg.eig(cov)

     #     # Transform the realizations
     #     transformed = np.dot(eigvectors, np.sqrt(np.diag(eigvalues))) @␣
      ↪realizations.T

     #     # Add the mean to each realization
     #     realizations = transformed.T + mean

         # Scatter plot
         plt.figure(figsize=(6,6))
```

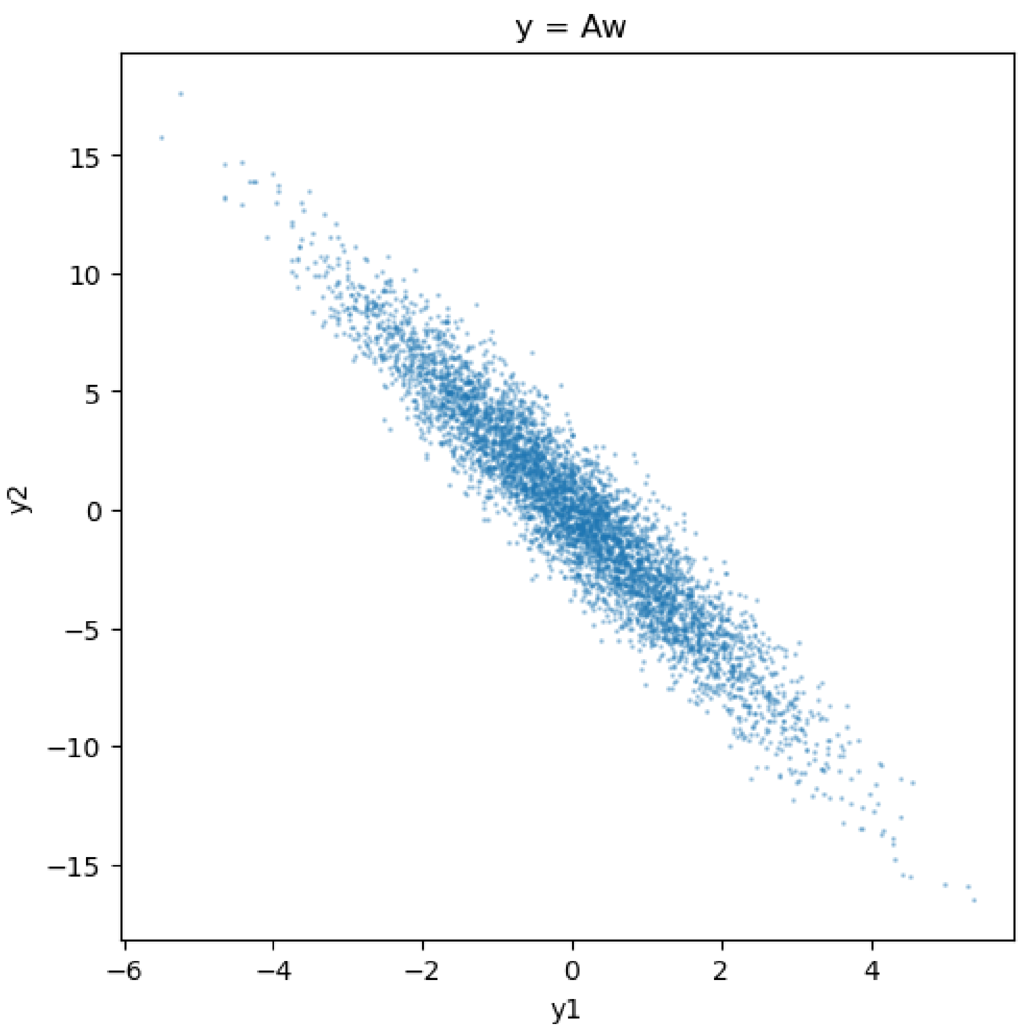
```
    plt.scatter(realizations[:, 0], realizations[:, 1], s=5, alpha=0.5)
    plt.title('5000 realizations of Gaussian random vector w')
    plt.xlabel('w1')
    plt.ylabel('w2')
    plt.show()
    return realizations
```

[5]:
```
mean_w = [0, 0]
cov_w = [[1, 0], [0, 1]]
w = white_Gaussian_vector(mean_w, cov_w, 5000)
```



[18]:
```
# Define the matrices A and w
A = np.array(([-1, -1], [2, 4]))

# w = np.array(w)
```

```python
# print(A.shape)
# print(w.shape)
y = np.matmul(w, A.T)
# print(y)

plt.figure(figsize=(6,6))
plt.scatter(y[:,0], y[:,1],s = 1, alpha=0.3)
plt.title(' y = Aw')
plt.xlabel('y1')
plt.ylabel('y2')
plt.show()

y_mean = np.mean(y, axis = 0)
N = y.shape[0]
cov_sample = 1/(N - 1) * np.dot((y - y_mean).T, y - y_mean)
cov_derived = A @ A.T

print(f"The sample covariance matrix is {cov_sample}")
print(f"The derived covariance matrix is {cov_derived}")
```

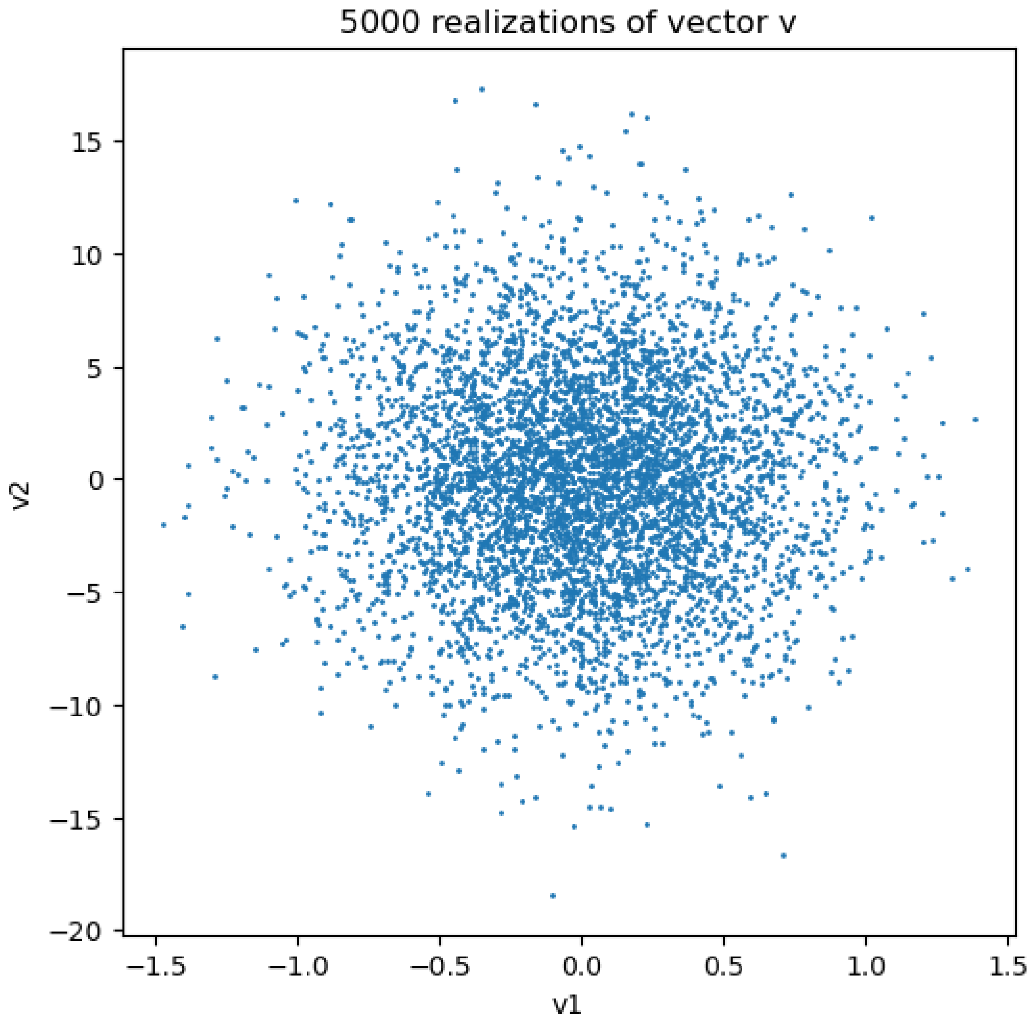



```
The sample covariance matrix is [[ 2.00033385 -5.99667272]
 [-5.99667272 19.96204433]]
The derived covariance matrix is [[ 2 -6]
 [-6 20]]
```

[40]:
```python
def decorrelate(y, cov_y):

    # Valculate eigenvalue and eigenvector
    eigvalues, eigvectors = np.linalg.eig(cov_y)

    lam = np.diag(eigvalues)
    E = eigvectors

#     print(E)
#     print(y)
```

```
    # v = E.T * y
#     v = np.dot(y, E)
    v = np.dot(E.T, y.T).T

    # Calculate covariance matrix of v
    cov_v = np.cov(v.T)

    # plot scatter
    plt.figure(figsize=(6,6))
    plt.scatter(v[:,0], v[:,1], s=1)
    plt.xlabel('v1')
    plt.ylabel('v2')
    plt.title('5000 realizations of vector v')
    plt.show()
    print(f'The covariance of v is {cov_v}')
    return lam, v
```

[41]:
```
lam_v, v = decorrelate(y, cov_derived)
```



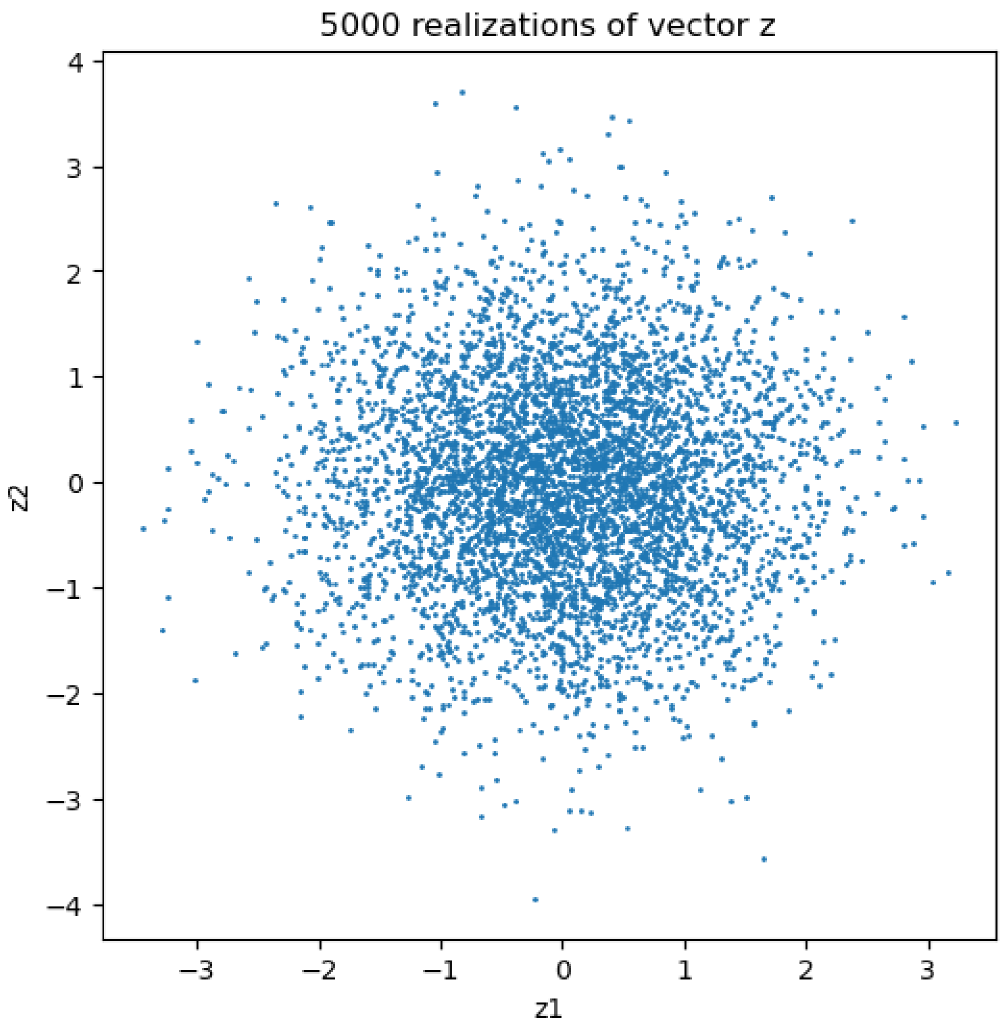

```
The covariance of v is [[ 1.82310308e-01 -7.85113202e-03]
 [-7.85113202e-03  2.17800679e+01]]
```

[42]:

```python
def whiten(v, lam):

    # Define z = lam^(-1/2)v
    lam_2 = np.linalg.inv(np.sqrt(lam))
#     print(lam_2.shape)
#     print(v.shape)
    z = np.dot(v, lam_2)

    # calculate covariance
    cov_z = np.cov(z.T)

    # plot scatter
```

```python
    plt.figure(figsize=(6,6))
    plt.scatter(z[:,0], z[:,1], s=1)
    plt.xlabel('z1')
    plt.ylabel('z2')
    plt.title('5000 realizations of vector z')
    plt.show()

    print(f'The covariance of z is {cov_z}')
```

[43]:
```python
whiten(v, lam)
```




```
The covariance of z is [[ 0.99435022 -0.00392557]
 [-0.00392557  0.99832303]]
```

[19]:
```python
def decorrelate(y, cov_y):
    '''
    This function decorrelates y and plot the scatter plot of v
    '''
    # calculate eigenvalue and eigenvector
    eigvalues, eigvectors = np.linalg.eig(cov_y)

    lam = np.diag(eigvalues)

    E = eigvectors

    # v = E.T * y
    v = np.dot(E.T, y)
#     v = np.dot(y, E)

    # calculate covariance
#     cov_v = np.cov(v.T)
    cov_v = np.cov(v)
    print(f'The covariance of v is {cov_v}')

    # plot scatter
    plt.figure(figsize=(6,6))
    plt.scatter(v[:,0], v[:,1], s=1)
    plt.xlabel('v1')
    plt.ylabel('v2')
    plt.title('5000 Realizations of V')
    plt.show()
    return lam, v
```

```
  File "/var/folders/qj/xxdr27w50r1735n7_r1yr4nm0000gn/T/ipykernel_10510/
↪880097702.py", line 26
    plt.title('5000 Realizations of V') plt.show()
                                        ^
SyntaxError: invalid syntax
```

[ ]:
```python
lam, v = decorrelate(y, cov_y)
```

hw8_3

April 17, 2023

[79]:
```
import numpy as np
import csv
import matplotlib.pyplot as plt
import random as rm
import sys
from sklearn import preprocessing
from sklearn.linear_model import Perceptron
from sklearn.model_selection import KFold
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
```

0.1 Get the data from csv file

[8]:
```
def getData(fname, dimension):
    # create a new array to store the data
    data = []
    label = []
    with open(fname, mode ='r')as file:
        # reading the CSV file
        csvFile = csv.reader(file)

        # displaying the contents of the CSV file
        for lines in csvFile:
            data.append(lines[1:])
            label.append(lines[0])
    data = np.array(data, dtype=float)
    label[0] = '1'
    label = np.array(label, dtype=float)
    return (data, label)
```

[9]:
```
xdata, ydata = getData("wine_data.csv",13)
# print(xdata)
```

[10]:
```
print(len(xdata))
print(len(ydata))
```

```
177
177
```

1 (a)Baseline for comparison

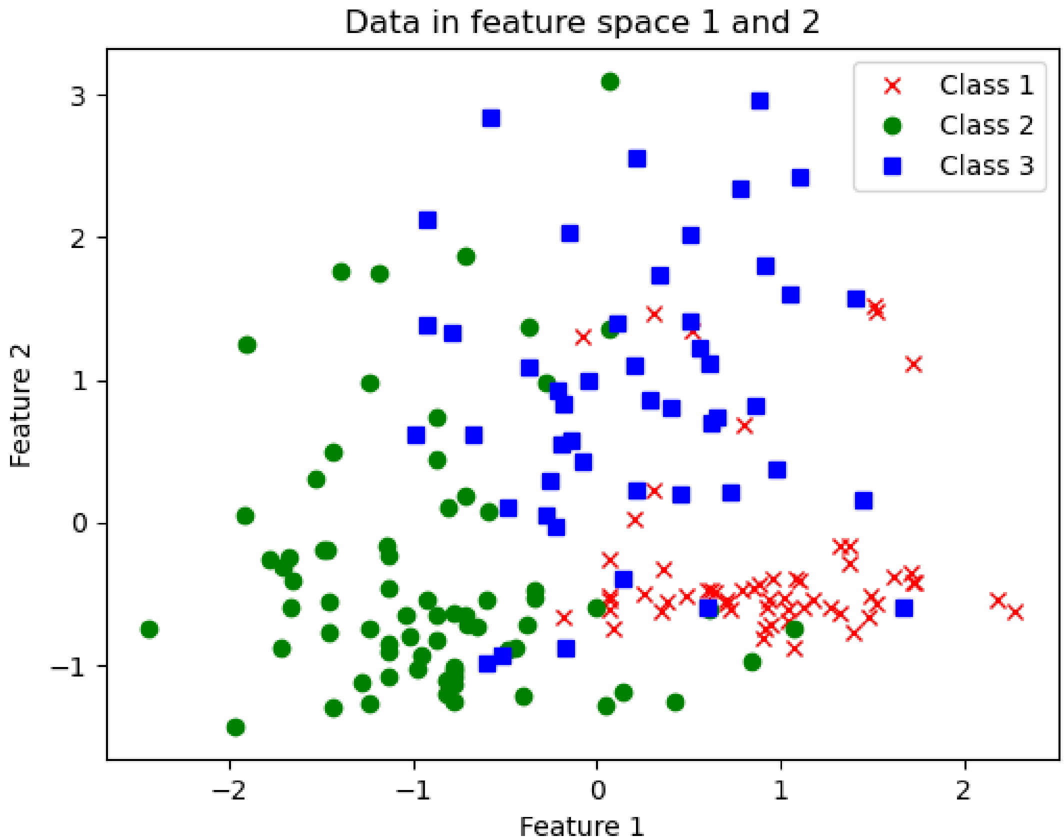
1.1 Standardize the dataset

[84]:
```
xdata_scalar = preprocessing.StandardScaler().fit(xdata)
xdata_standard = xdata_scalar.transform(xdata)
```

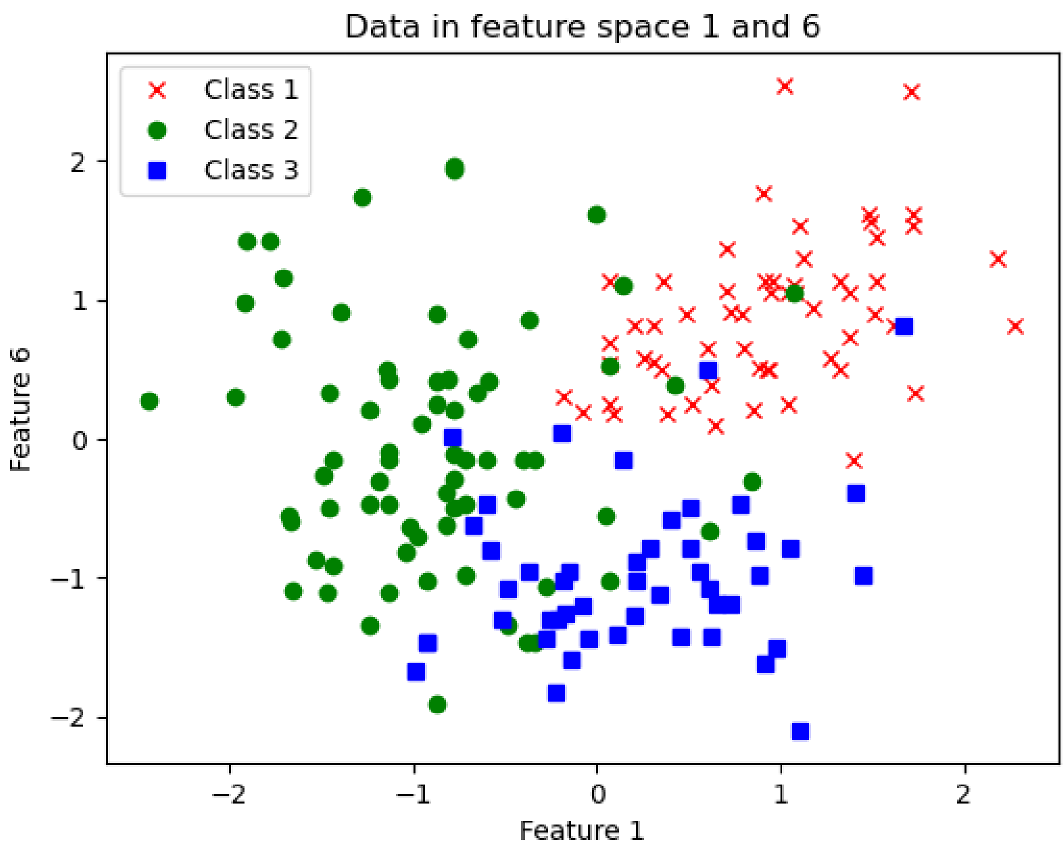
1.1.1 Plot different features

[12]:
```
def plot_features(xdata, ydata, feature1, feature2):

    plt.title(f"Data in feature space {feature1 + 1} and {feature2 + 1}")
    plt.plot(xdata[ydata == 1, feature1],xdata[ydata == 1, feature2], 'rx',␣
 ↪label = "Class 1")
    plt.plot(xdata[ydata == 2, feature1],xdata[ydata == 2, feature2], 'go',␣
 ↪label = "Class 2")
    plt.plot(xdata[ydata == 3, feature1],xdata[ydata == 3, feature2], 'bs',␣
 ↪label = "Class 3")
    plt.xlabel(f"Feature {feature1 + 1}")
    plt.ylabel(f"Feature {feature2 + 1}")
    plt.legend()
    plt.show()
```

[13]:
```
# plot the data projected into x1, x2 space
plot_features(xdata_standard, ydata, 0, 1)
```



[8]:
```
# plot the data projected into x1, x6 space
plot_features(xdata_standard, ydata, 0, 5)
```



1.2 Run a multiclass perceptron classifier on the 2D data

[14]:
```
def shuffle(data, label):
    newData = np.copy(data)
    newLabel = np.copy(label)
    N = len(newData)
    shuff = np.random.permutation(N)
    for i in range(N):
        newData[i] = data[shuff[i]]
        newLabel[i] = label[shuff[i]]
#     print(newData)
    return (newData, newLabel)
```

[15]:
```
def MLP_classifier(xdata_orig, ydata_orig):
    xdata = np.copy(xdata_orig)
    ydata = np.copy(ydata_orig)

    # Run 5 times
    weight_first_fold = []
```

```
    mean_error_rate = []
    for r in range(5):
        xdata, ydata = shuffle(xdata, ydata)

        # Define the cross-validation object
        error_rate_history = []
        cv = KFold(n_splits=20)
        for i, (train_index, val_index) in enumerate(cv.split(xdata)): # i in␣
↪range of 20
            train_xdata = xdata[train_index]
            train_ydata = ydata[train_index]
            val_xdata = xdata[val_index]
            val_ydata = ydata[val_index]

            mlp = Perceptron(fit_intercept = False)
            mlp.fit(train_xdata, train_ydata)
            error_rate_history.append(1 - mlp.score(val_xdata, val_ydata))
            if( i == 0 ):
                weight_first_fold.append(mlp.coef_)
        print(f"The mean classification error rate in run {r + 1} = {np.
↪mean(error_rate_history)}")
        mean_error_rate.append(np.mean(error_rate_history))
    print(f"The average and standard deviation of the mean classification error␣
↪over the 5 runs is {np.mean(mean_error_rate)} and {np.std(mean_error_rate)}")
    return weight_first_fold
```

[61]:
```
################################################
## EE559 HW1, Prof. Jenkins
## Created by Arindam Jati
## Modified by Lei Lei
################################################

import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist


def plotDecBoundaries(training, label_train, w, title, inc = 0.01):

    # Plot the decision boundaries and data points for perceptron learning␣
↪classification result
    # training: traning data
    # label_train: class lables correspond to training data
    # w: weight vector
    # title: the title of the plot
    # inc: step size
    nclass = max(np.unique(label_train))
```

```
    # Set the feature range for ploting
    max_x = np.ceil(max(training[:, 0])) + 1
    min_x = np.floor(min(training[:, 0])) - 1
    max_y = np.ceil(max(training[:, 1])) + 1
    min_y = np.floor(min(training[:, 1])) - 1

    xrange = (min_x, max_x)
    yrange = (min_y, max_y)

    # step size for how finely you want to visualize the decision boundary.
#     inc = 0.5
    inc = inc

    # generate grid coordinates. this will be the basis of the decision
    # boundary visualization.
#     (x, y) = np.meshgrid(np.arange(xrange[0], xrange[1] + inc / 100, inc),
#                          np.arange(yrange[0], yrange[1] + inc / 100, inc))

    (x, y) = np.meshgrid(np.arange(xrange[0], xrange[1] + inc / 100,  inc),
                         np.arange(yrange[0], yrange[1] + inc / 100,  inc))

    xy = np.hstack((x.reshape(x.shape[0] * x.shape[1], 1, order='F'),
                    y.reshape(y.shape[0] * y.shape[1], 1, order='F')))  # make␣
 ↪(x,y) pairs as a bunch of row vectors.


    pred_label = np.zeros(np.shape(xy)[0])
    for i in range(np.shape(xy)[0]):
        pred_label[i] = np.argmax(np.dot(w, xy[i].T))

    # size of the (x, y) image, which will also be the size of the
    # decision boundary image that is used as the plot background.
    image_size = x.shape
#     print(image_size)



    decisionmap = pred_label.reshape(image_size, order='F')
#     plt.contour(x, y, decisionmap, colors='k', levels=[0, 1, 2])
#     plt.contourf(x, y, decisionmap, cmap=plt.cm.Set1, alpha=0.5)

    plt.imshow(decisionmap, extent=[xrange[0], xrange[1], yrange[0],␣
 ↪yrange[1]], origin='lower',aspect='auto')
#     plt.imshow(decisionmap, aspect='auto')
```

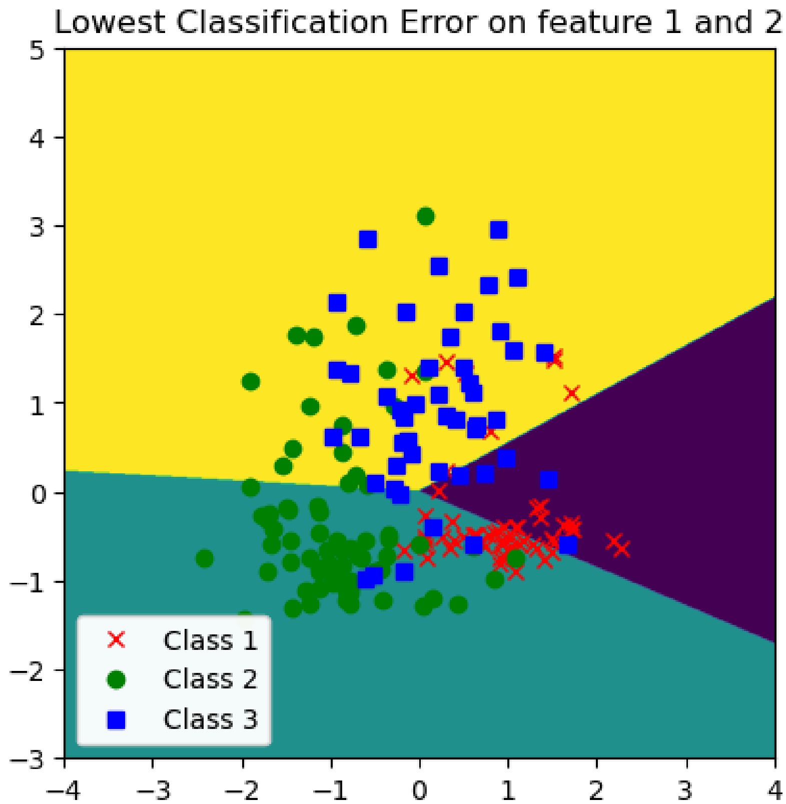
```
    # plot the class training data.
    plt.plot(training[label_train == 1, 0], training[label_train == 1, 1], 'rx')
    plt.plot(training[label_train == 2, 0], training[label_train == 2, 1], 'go')
    plt.plot(training[label_train == 3, 0], training[label_train == 3, 1], 'bs')

    plt.title(title)
    l = plt.legend(('Class 1', 'Class 2', 'Class 3'), loc=3)
#     plt.gca().add_artist(l)

    # plot the class mean vector.
    plt.show()
```

Run a multiclass perceptron classifier on the 2D data using only features x1, x2

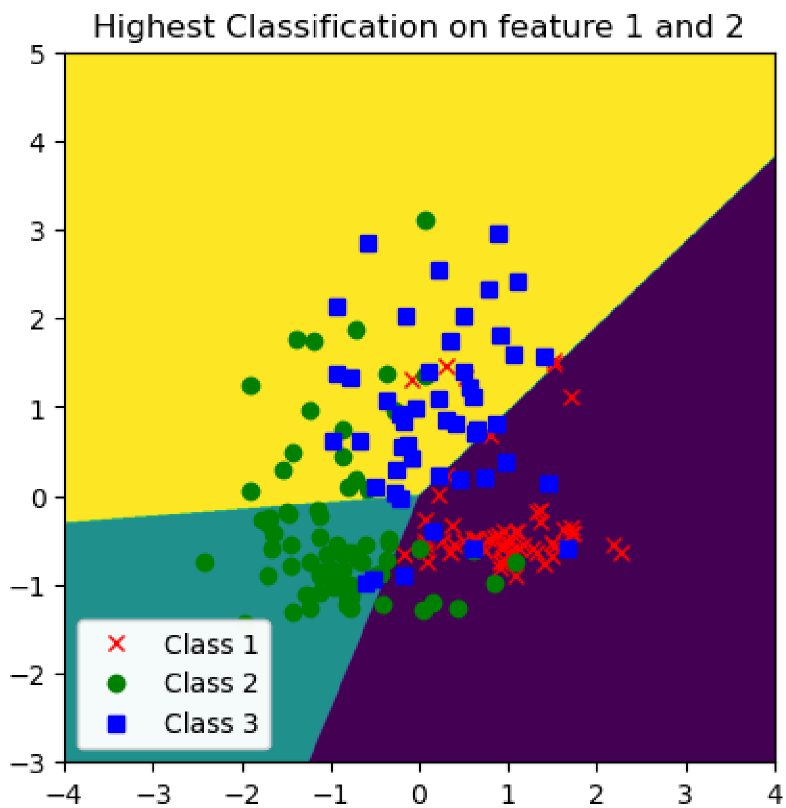
[129]:
```
weight_12 = MLP_classifier(xdata_standard[:,0:2], ydata)
weight_12 = np.array(weight_12)
```

```
The mean classification error rate in run 1 = 0.28125000000000006
The mean classification error rate in run 2 = 0.25972222222222224
The mean classification error rate in run 3 = 0.23680555555555557
The mean classification error rate in run 4 = 0.225
The mean classification error rate in run 5 = 0.31527777777777778
The average and standard deviation of the mean classification error over the 5
runs is 0.26361111111111113 and 0.03226469901407941
```

[130]:
```
plotDecBoundaries(xdata_standard[:,0:2], ydata, weight_12[3], 'Lowest␣
 ↪Classification Error on feature 1 and 2' )
```



[131]:
```
plotDecBoundaries(xdata_standard[:,0:2], ydata, weight_12[4], 'Highest␣
↪Classification on feature 1 and 2' )
```

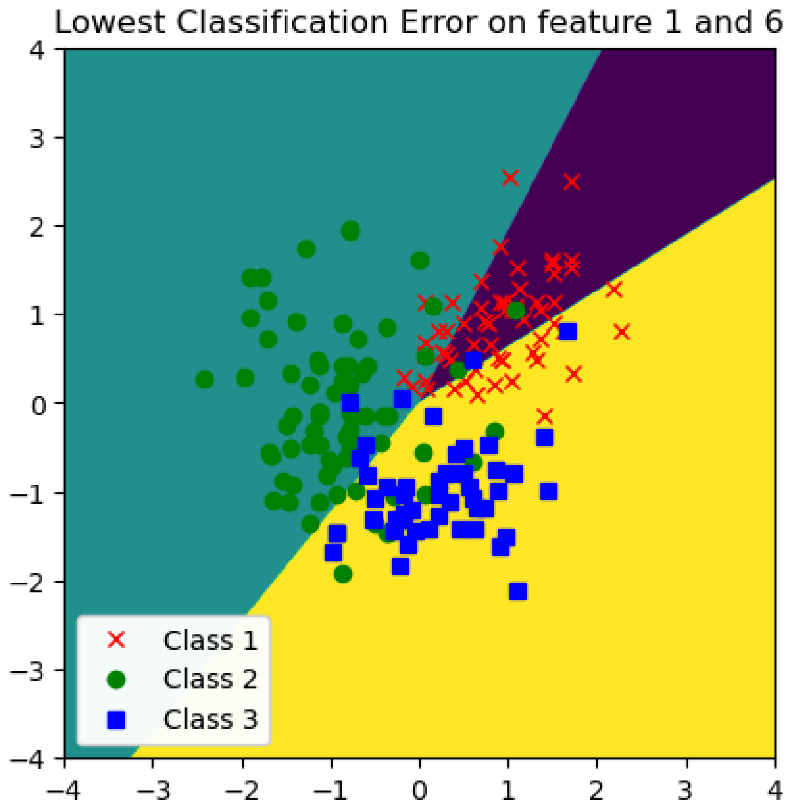


Run a multiclass perceptron classifier on the 2D data using only features x1, x6

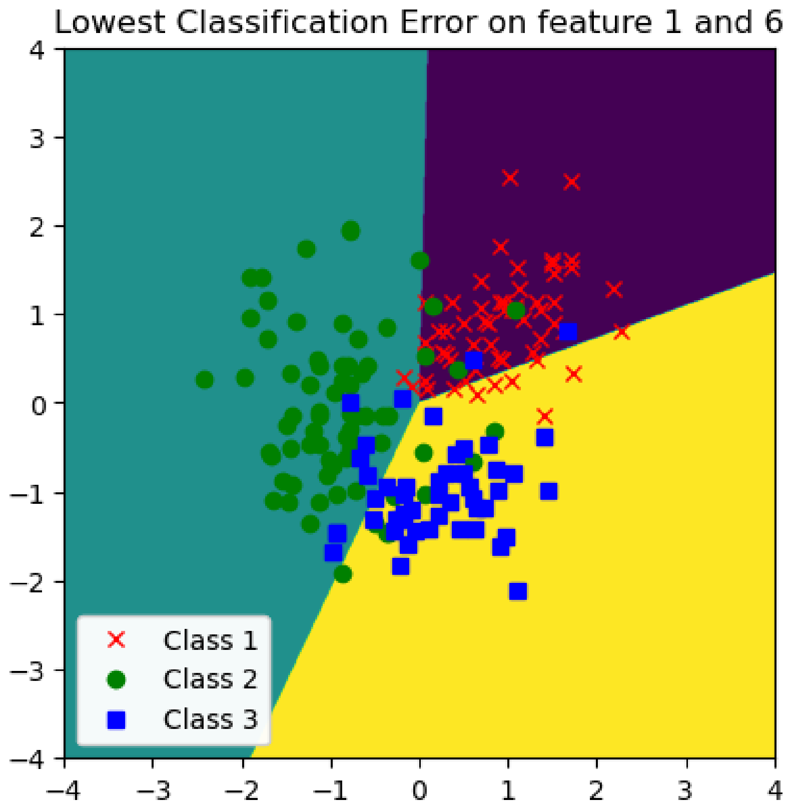
[136]:
```
weight_16 = MLP_classifier(xdata_standard[:,[0,5]], ydata)
weight_16 = np.array(weight_16)
```

```
The mean classification error rate in run 1 = 0.24930555555555559
The mean classification error rate in run 2 = 0.23819444444444446
The mean classification error rate in run 3 = 0.22083333333333335
The mean classification error rate in run 4 = 0.2534722222222222
The mean classification error rate in run 5 = 0.22916666666666666
The average and standard deviation of the mean classification error over the 5
runs is 0.23819444444444446 and 0.012163685581006285
```

[139]:
```
plotDecBoundaries(xdata_standard[:,[0,5]], ydata, weight_16[2], 'Lowest␣
↪Classification Error on feature 1 and 6' )
```



[140]:
```
plotDecBoundaries(xdata_standard[:,[0,5]], ydata, weight_16[3], 'Lowest␣
↪Classification Error on feature 1 and 6' )
```

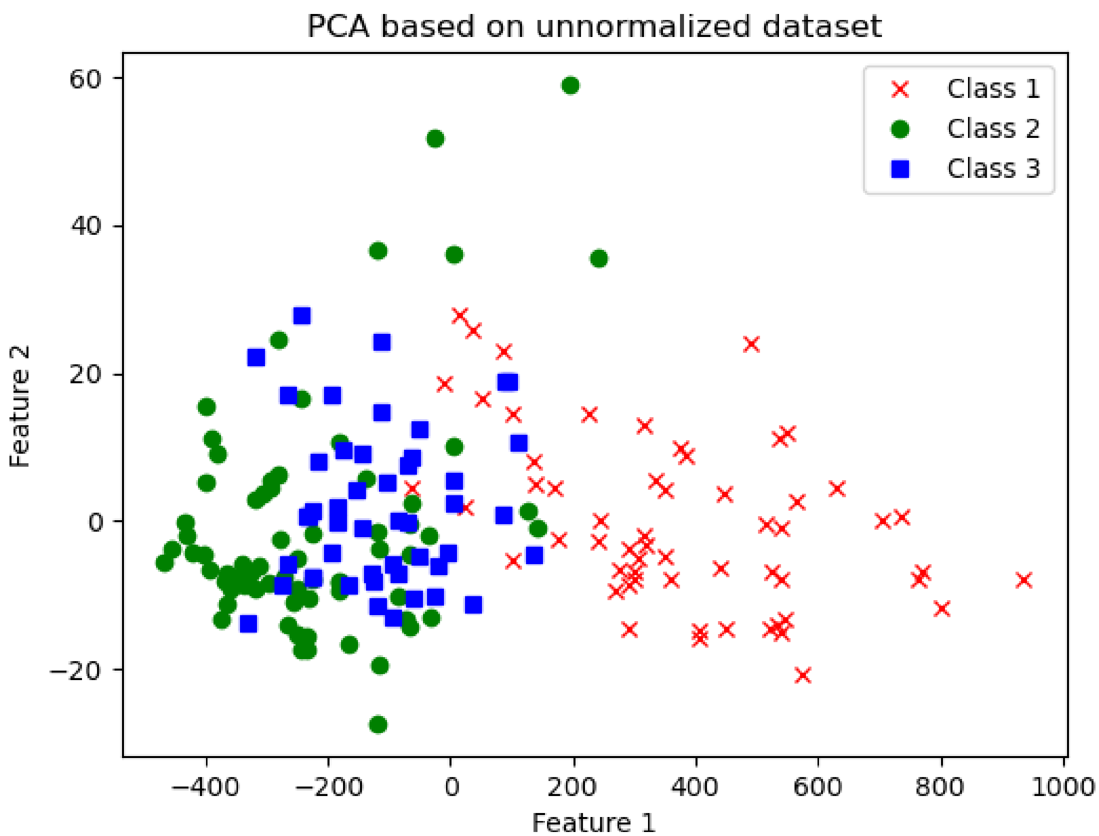


2 (b) PCA based on unnormalized dataset.

[70]:
```python
def plot_PCA(xdata_orig, ydata_orig, title):

    xdata_ = np.copy(xdata_orig)
    ydata = np.copy(ydata_orig)

    pca = PCA(n_components=2)
#     xdata = pca.fit(xdata).transform(xdata)
    xdata = pca.fit_transform(xdata_)

    plt.title(title)
    plt.plot(xdata[ydata == 1, 0],xdata[ydata == 1, 1], 'rx', label = "Class 1")
    plt.plot(xdata[ydata == 2, 0],xdata[ydata == 2, 1], 'go', label = "Class 2")
    plt.plot(xdata[ydata == 3, 0],xdata[ydata == 3, 1], 'bs', label = "Class 3")
    plt.xlabel(f"Feature 1")
    plt.ylabel(f"Feature 2")
    plt.legend()
    plt.show()
```

```
    return xdata
```

[71]:
```
xdata_pca_unnormal = plot_PCA(xdata, ydata, "PCA based on unnormalized dataset")
```

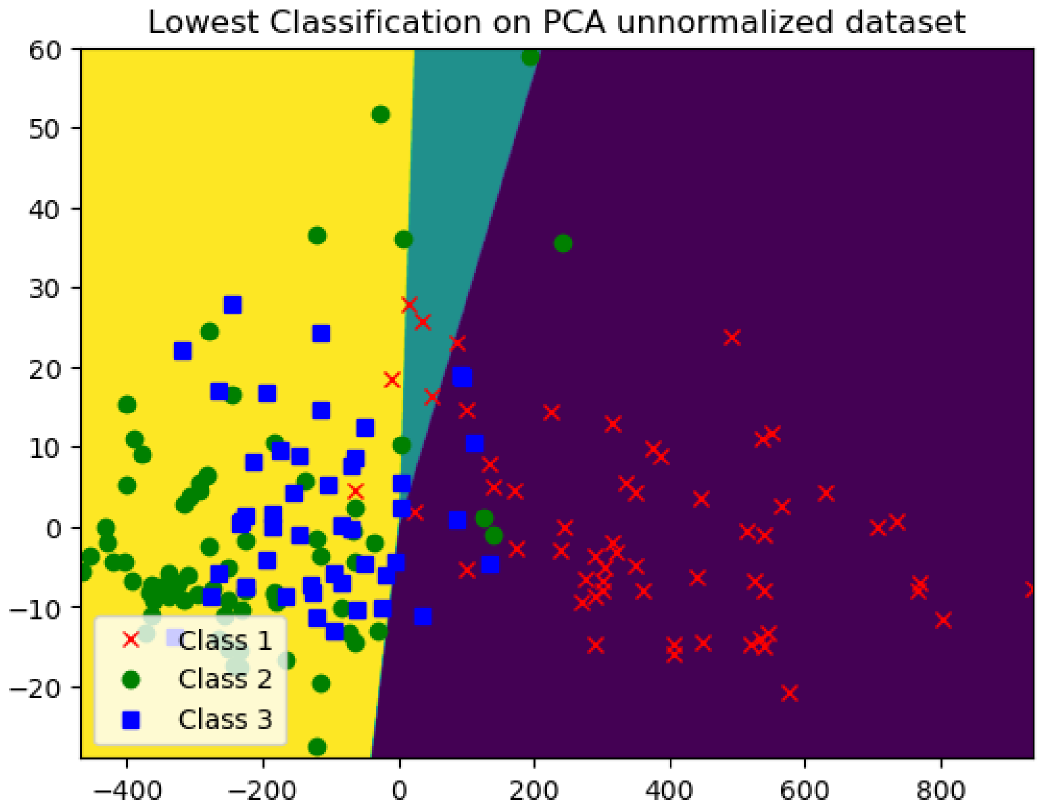


[30]:
```
weights_pca_unnormal = MLP_classifier(xdata_pca_unnormal, ydata)
```

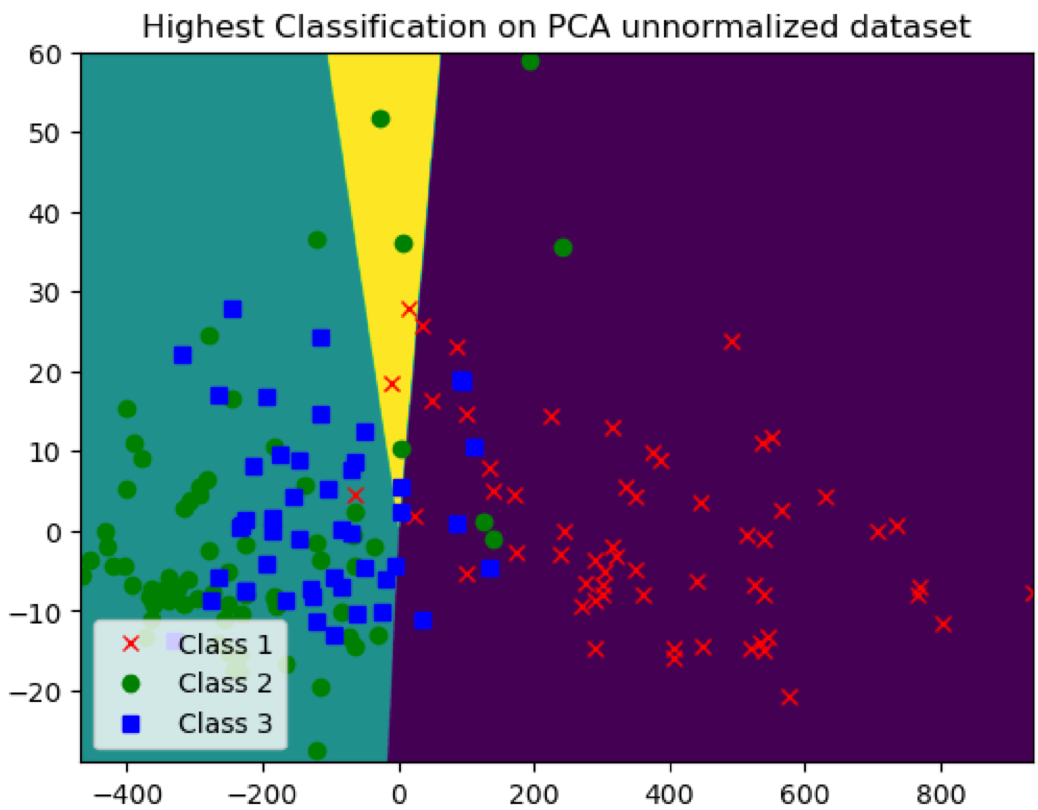
```
The mean classification error rate in run 1 = 0.34375000000000006
The mean classification error rate in run 2 = 0.36944444444444446
The mean classification error rate in run 3 = 0.46944444444444444
The mean classification error rate in run 4 = 0.4583333333333333
The mean classification error rate in run 5 = 0.45138888888888895
The average and standard deviation of the mean classification error over the 5
runs is 0.41847222222222225 and 0.051493137502744396
```

[60]:
```
plotDecBoundaries(xdata_pca_unnormal, ydata, weights_pca_unnormal[0], 'Lowest␣
 ↪Classification on PCA unnormalized dataset',inc = 0.5 )
```

```
(179, 2811)
```

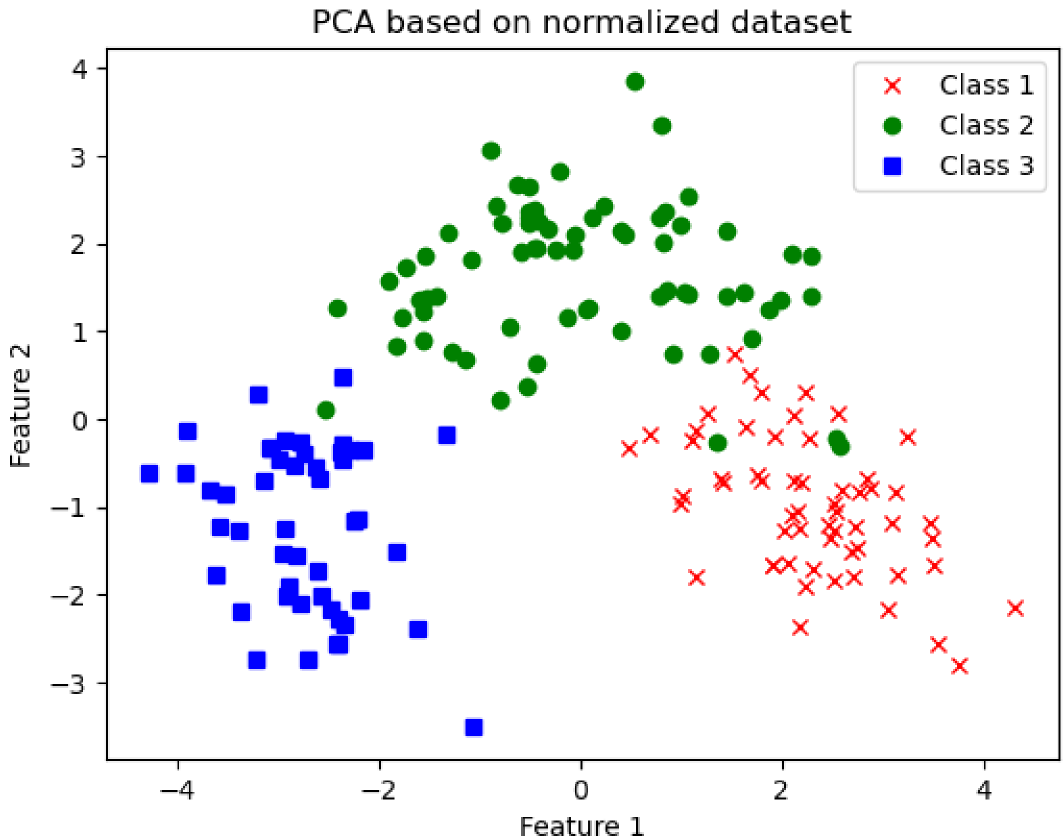


[62]:
```
plotDecBoundaries(xdata_pca_unnormal, ydata, weights_pca_unnormal[2], 'Highest␣
↪Classification on PCA unnormalized dataset',inc = 0.5 )
```



2.1 (c) PCA based on standardized dataset.

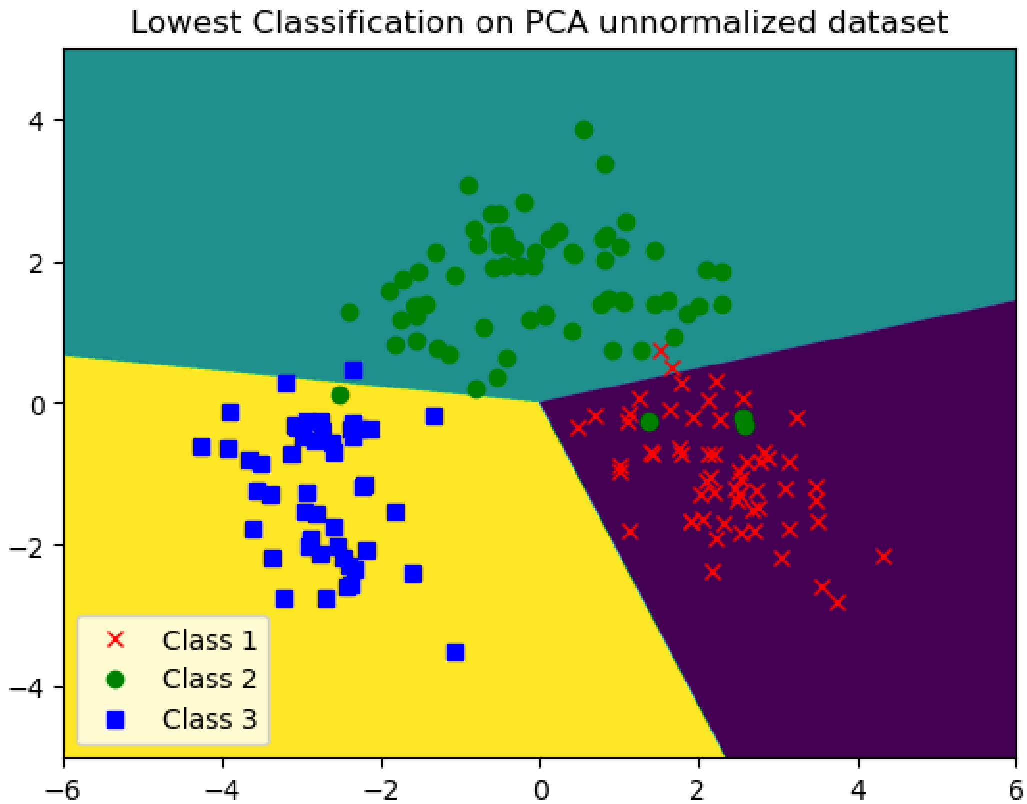
[72]:
```
xdata_pca_normal = plot_PCA(xdata_standard, ydata, "PCA based on normalized␣
↪dataset")
```



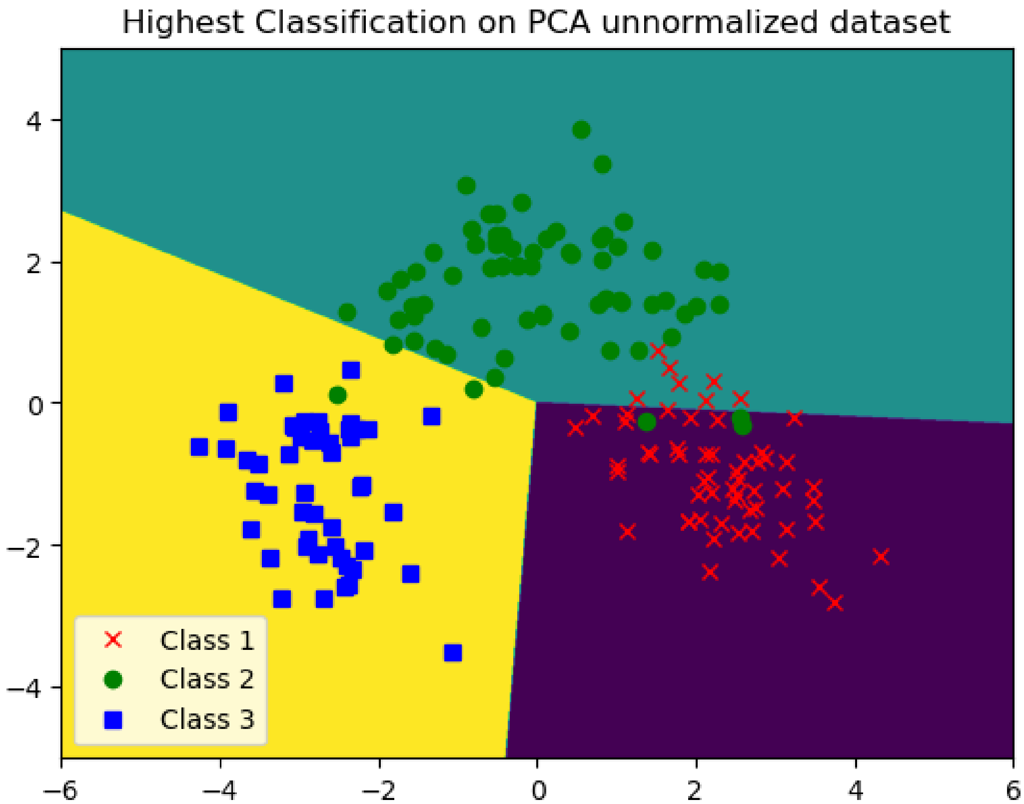
[73]:
```
weights_pca_normal = MLP_classifier(xdata_pca_normal, ydata)
```

```
The mean classification error rate in run 1 = 0.05208333333333335
The mean classification error rate in run 2 = 0.052083333333333336
The mean classification error rate in run 3 = 0.0388888888888889
The mean classification error rate in run 4 = 0.04444444444444445
The mean classification error rate in run 5 = 0.10277777777777779
The average and standard deviation of the mean classification error over the 5
runs is 0.05805555555555557 and 0.022908668637994484
```

[77]:
```
plotDecBoundaries(xdata_pca_normal, ydata, weights_pca_normal[2], 'Lowest␣
  ↪Classification on PCA unnormalized dataset',inc = 0.01 )
```



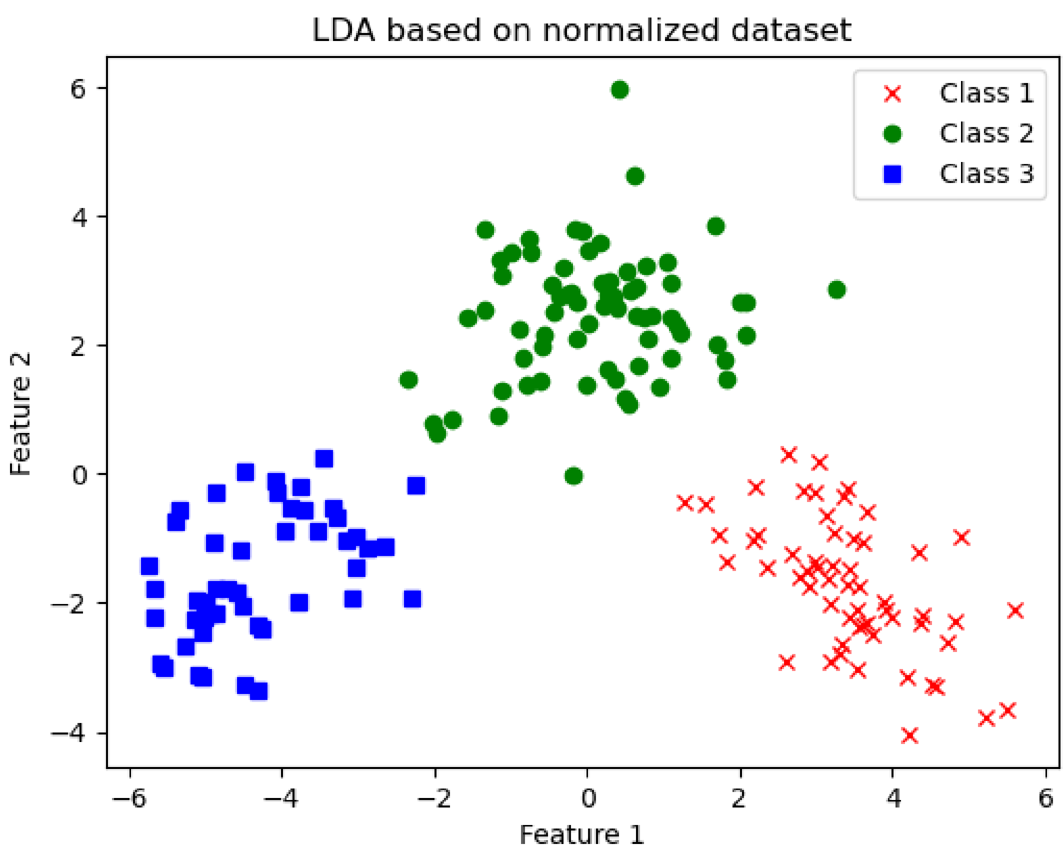

```
[92]: plotDecBoundaries(xdata_pca_normal, ydata, weights_pca_normal[4], 'Highest␣
      ↪Classification on PCA unnormalized dataset',inc = 0.01 )
```



2.2 (d) MDA (using LDA as an approximation to MDA).

[82]:
```python
def plot_MDA(xdata_orig, ydata_orig, title):

    xdata_ = np.copy(xdata_orig)
    ydata = np.copy(ydata_orig)

    lda = LinearDiscriminantAnalysis(n_components=2)
#     xdata = pca.fit(xdata).transform(xdata)
    xdata = lda.fit_transform(xdata_, ydata)

    plt.title(title)
    plt.plot(xdata[ydata == 1, 0],xdata[ydata == 1, 1], 'rx', label = "Class 1")
    plt.plot(xdata[ydata == 2, 0],xdata[ydata == 2, 1], 'go', label = "Class 2")
    plt.plot(xdata[ydata == 3, 0],xdata[ydata == 3, 1], 'bs', label = "Class 3")
    plt.xlabel(f"Feature 1")
    plt.ylabel(f"Feature 2")
    plt.legend()
    plt.show()

    return xdata
```

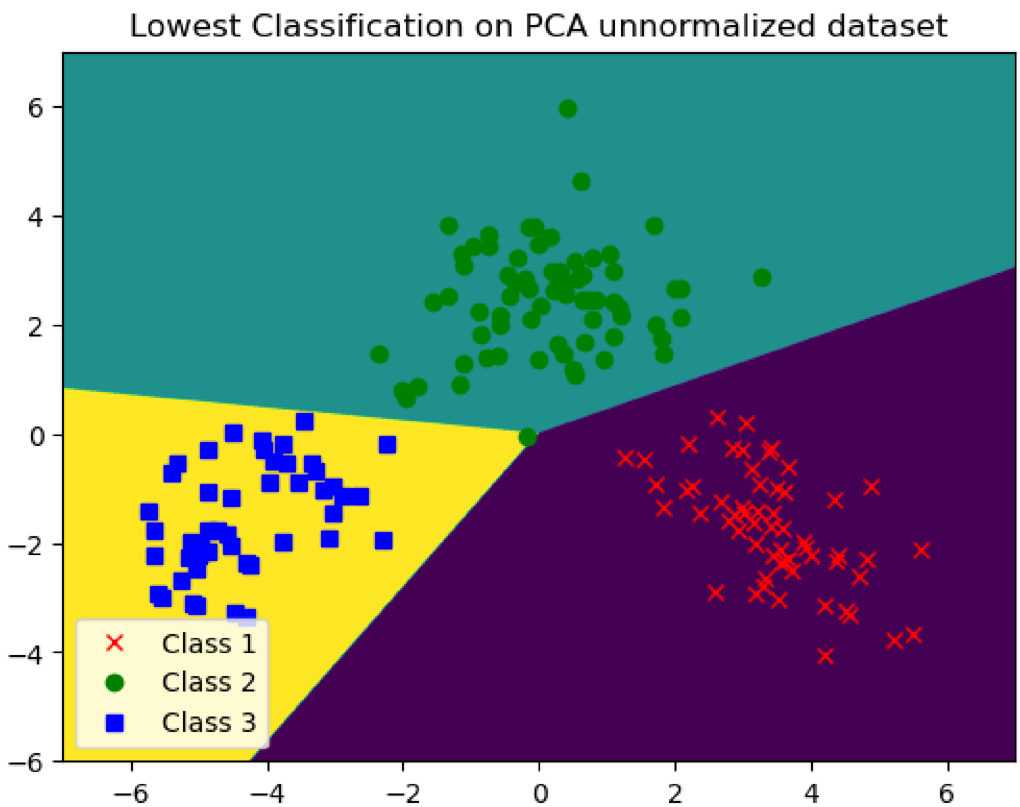
[87]:
```python
xdata_lda_normal = plot_MDA(xdata_standard, ydata, "LDA based on normalized␣
↪dataset")
```



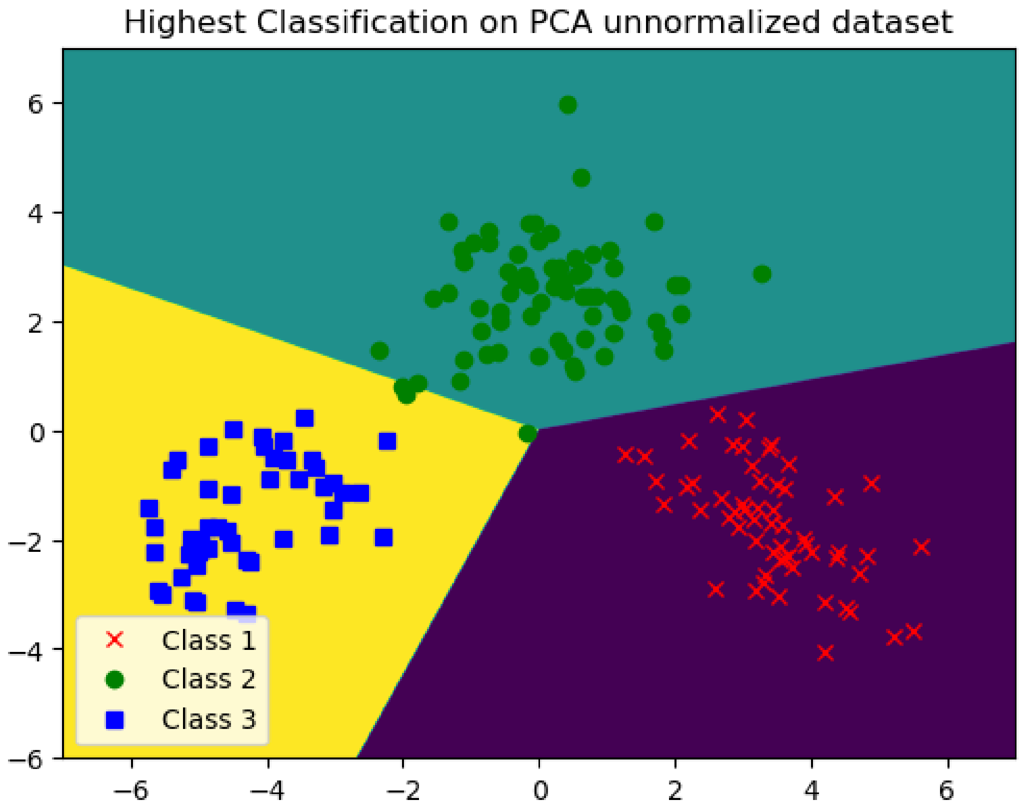
[88]:
```
weights_lda_normal = MLP_classifier(xdata_lda_normal, ydata)
```

```
The mean classification error rate in run 1 = 0.01666666666666667
The mean classification error rate in run 2 = 0.011111111111111117
The mean classification error rate in run 3 = 0.005555555555555558
The mean classification error rate in run 4 = 0.016666666666666673
The mean classification error rate in run 5 = 0.011111111111111117
The average and standard deviation of the mean classification error over the 5
runs is 0.012222222222222226 and 0.004157397096415491
```

[89]:
```
plotDecBoundaries(xdata_lda_normal, ydata, weights_lda_normal[2], 'Lowest␣
↪Classification on PCA unnormalized dataset',inc = 0.01 )
```



[91]:
```
plotDecBoundaries(xdata_lda_normal, ydata, weights_lda_normal[0], 'Highest␣
↪Classification on PCA unnormalized dataset',inc = 0.01 )
```


Highest Classification on PCA unnormalized dataset
Class 1
Class 2
Class 3
6
4
2
0
−2
−4
−6
−6
−4
−2
0
2
4
6

[ ]:

hw8_4

April 17, 2023

[12]:
```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
```

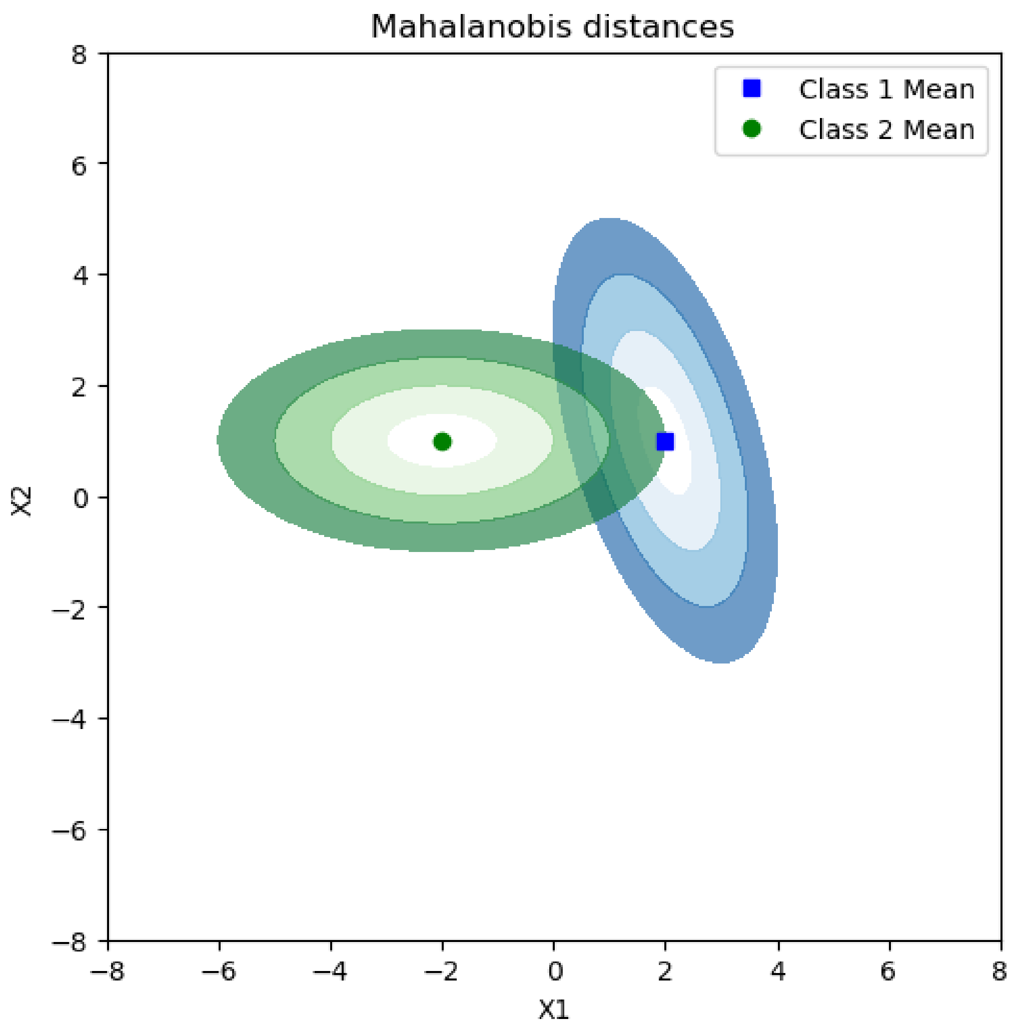
[13]:
```
# mean vectors
m1 = np.array([2, 1])
m2 = np.array([-2, 1])

# covariance matrices
cov1 = np.array([[1, -1], [-1, 4]])
cov2 = np.array([[4, 0], [0, 1]])

# values of B
B_range = [0.5, 1.0, 1.5, 2.0]
```

[43]:
```
# define the range of x and y values
x = np.linspace(-8, 8, 1000)
y = np.linspace(-8, 8, 1000)

# create a meshgrid from x and y values
X, Y = np.meshgrid(x, y)
Z = np.column_stack([X.ravel(), Y.ravel()])
```

[44]:
```
def mahalanobis_dist(point, mean, cov, coordinate):
    """
    Calculate the Mahalanobis distance between a point and a mean vector
    with covariance matrix cov.
    """
    inv_cov = np.linalg.inv(cov)
    diff = point - mean
    dis = np.sqrt(np.sum(diff @ inv_cov * diff, axis=1)).reshape(coordinate.
 ↪shape)
    return dis
```

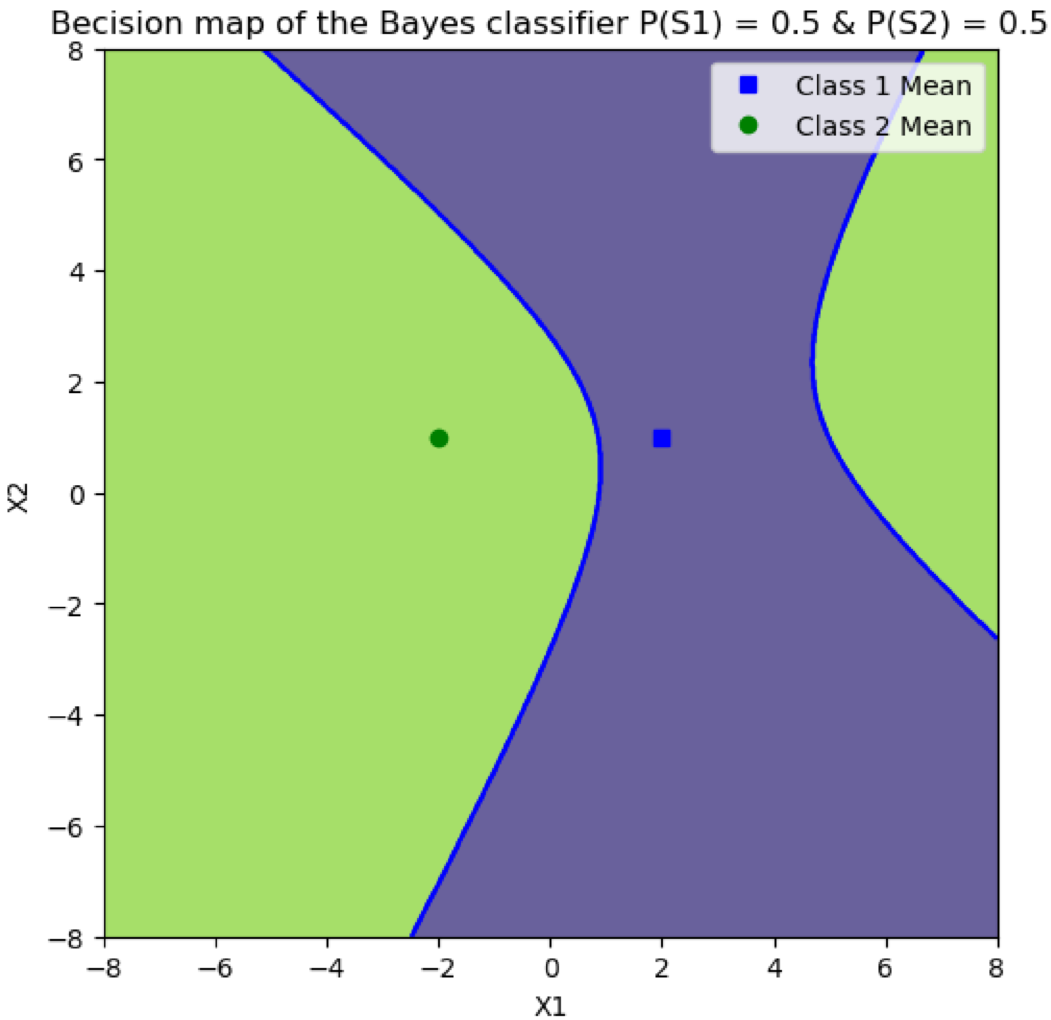
[47]:
```
# define the values of B
B_values = [0.5, 1.0, 1.5, 2.0]
```

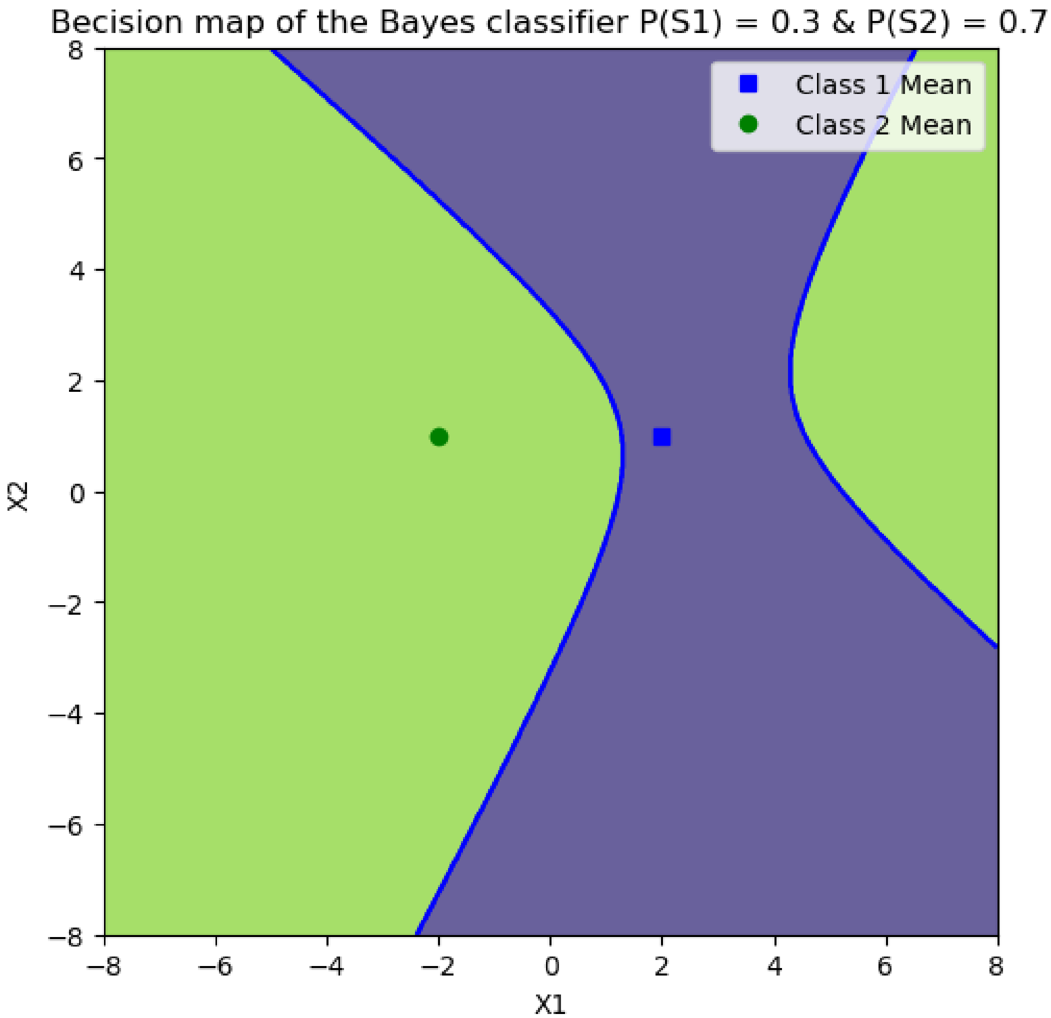
```python
# plot the distance
d1 = mahalanobis_dist(Z, m1, cov1, X)
d2 = mahalanobis_dist(Z, m2, cov2, Y)

# plot the mean points
plt.figure(figsize=(6, 6))
plt.plot(m1[0], m1[1], 'bs', label='Class 1 Mean')
plt.plot(m2[0], m2[1], 'go', label='Class 2 Mean')

# plot Mahalanobis distances
plt.contourf(X, Y, d1, alpha=0.6, cmap='Blues', levels=B_range)
plt.contourf(X, Y, d2, alpha=0.6, cmap='Greens', levels=B_range)

plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Mahalanobis distances')
plt.legend()
plt.show()
```

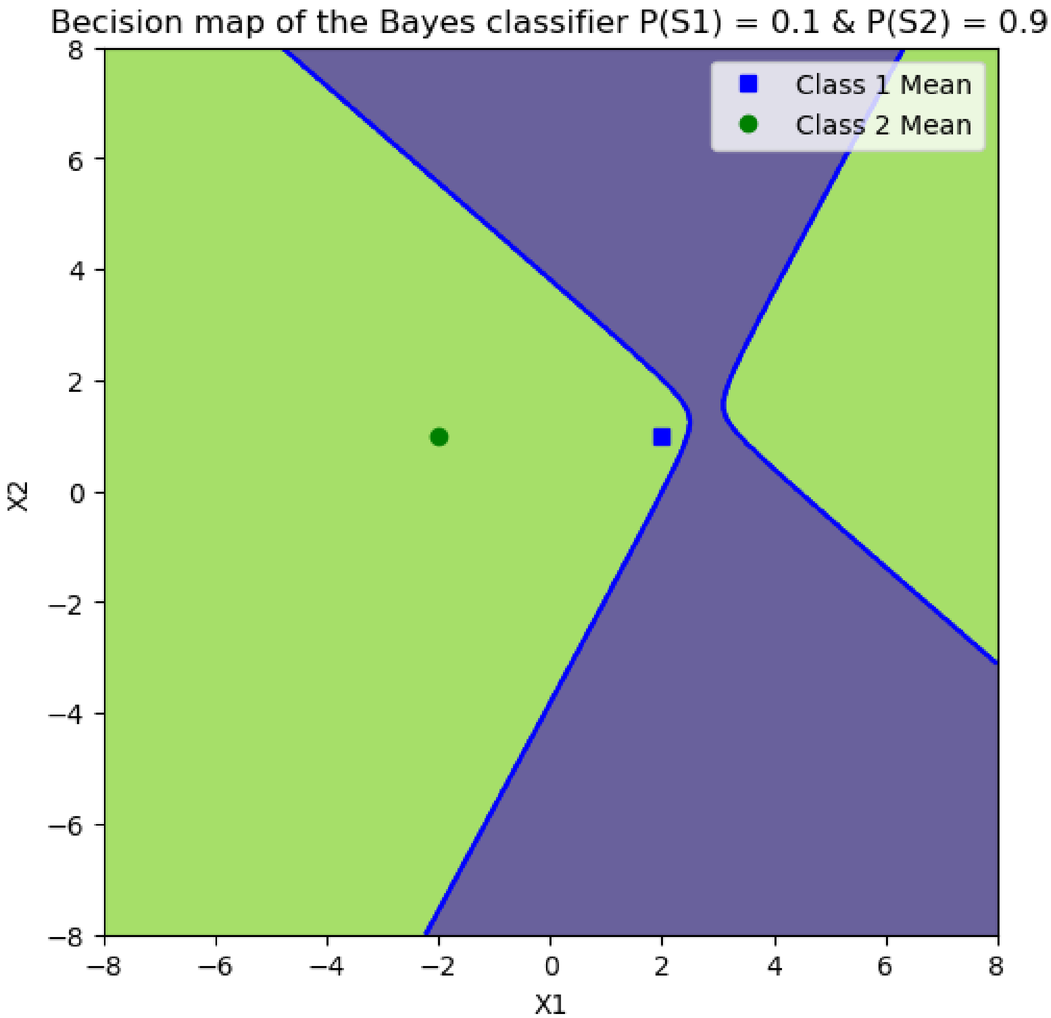


[55]:
```python
def plot_Bayes(s1, s2):
    # calculate Mahalanobis distances
    d1_baye = np.sum((Z - m1) @ np.linalg.inv(cov1) * (Z - m1), axis=1)
    d2_baye = np.sum((Z - m2) @ np.linalg.inv(cov2) * (Z - m2), axis=1)


    # decision boundary
    decisionmap = d1_baye - d2_baye - 2 * np.log(s1 / s2) - np.log(np.linalg.
 ↪det(cov1) / np.linalg.det(cov2))

    XY = np.where(decisionmap >= 0, 2, 1).reshape(X.shape)
    # plot the mean points
    plt.figure(figsize=(6, 6))
    plt.plot(m1[0], m1[1], 'bs', label='Class 1 Mean')
    plt.plot(m2[0], m2[1], 'go', label='Class 2 Mean')
```

```
    # plot decision boundary
    plt.contour(X, Y, XY, colors='blue', levels=B_range)
    plt.contourf(X, Y, XY, alpha=0.8, levels=B_range)
    plt.xlabel('X1')
    plt.ylabel('X2')
    plt.title(f'Becision map of the Bayes classifier P(S1) = {s1} & P(S2) =␣
 ↪{s2}')
    plt.legend()
    plt.show()

plot_Bayes(0.5, 0.5)
plot_Bayes(0.3, 0.7)
plot_Bayes(0.1, 0.9)
```




Becision map of the Bayes classifier P(S1) = 0.3 & P(S2) = 0.7
Class 1 Mean
Class 2 Mean
X1
X2
−8
−6
−4
−2
0
2
4
6
8


Becision map of the Bayes classifier P(S1) = 0.1 & P(S2) = 0.9
Class 1 Mean
Class 2 Mean
X1
X2
−8
−6
−4
−2
0
2
4
6
8

```
[ ]:
```