Question 5.1

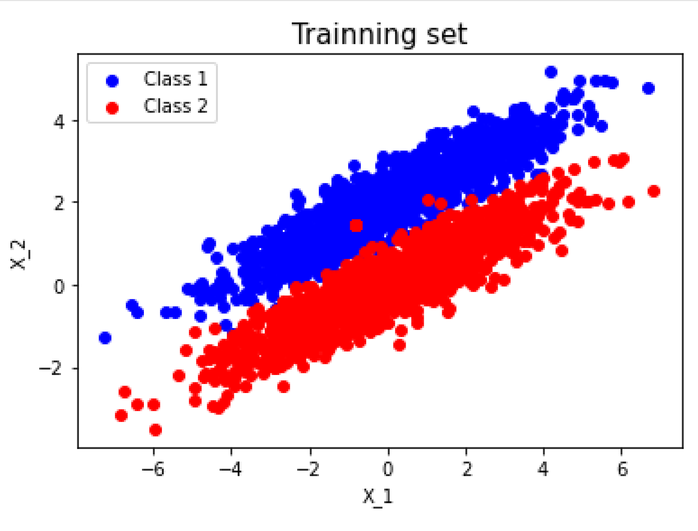
29]:

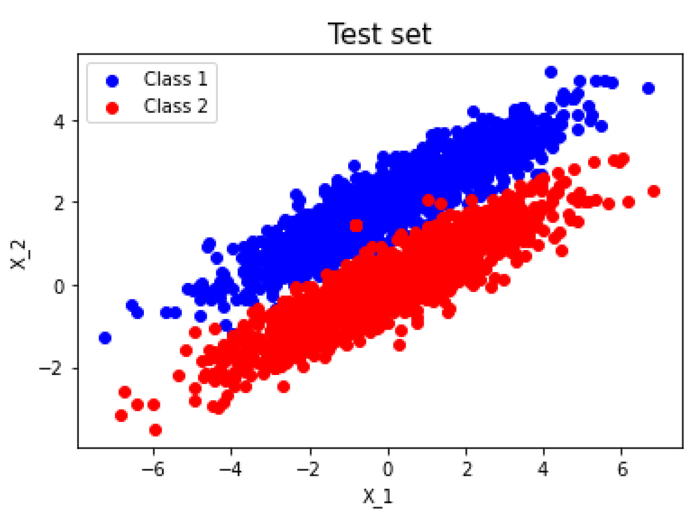
```python
import numpy as np
import math
from functools import reduce
from matplotlib import pyplot as plt

def multivariate_normal_pdf(x, mean, sigma):
    l = x.shape[0]
    det_S = np.linalg.det(sigma)
    norm_const = 1.0/((2.0*np.pi)**(l/2.0)*np.sqrt(det_S))
    inv_S = np.linalg.inv(sigma)
    a1 = np.dot(np.dot((x-mean), inv_S), (x-mean))

    return norm_const*np.exp(-(1.0/2.0)*a1)

def multivariate_normal_pdf_v2(x, mean, sigma):
    l = x.shape[1]
    det_S = np.linalg.det(sigma)
    norm_const = 1.0/((2.0*np.pi)**(l/2.0)*np.sqrt(det_S))
    inv_S = np.linalg.inv(sigma)
    a1    = np.sum(np.dot(x-mean, inv_S)*(x-mean), axis = 1)

    return norm_const*np.exp(-0.5*a1)
```

i

30]:

```python
#Generation and plotting of trainning and testing sets

N = 1500 # Number of data points per class
m1 = np.array([0, 2]) #mean for first class
m2 = np.array([0, 0]) #mean for second class
S = np.array([[4, 1.8], [1.8, 1]]) #covariance matrix
p = 2*N

X_class1 = np.random.multivariate_normal(m1,S,N)
X_class2 = np.random.multivariate_normal(m2, S,N)

X_1 = np.concatenate((X_class1, X_class2), axis = 0) #data_set
Y_1 = np.concatenate((np.zeros((N, 1)), 1*np.ones((N, 1))), axis = 0)

plt.figure(1)
plt.scatter(X_1[np.nonzero(Y_1 == 0),0], X_1[np.nonzero(Y_1 == 0),1], color = "b",label = "Class 1")
plt.scatter(X_1[np.nonzero(Y_1 == 1),0], X_1[np.nonzero(Y_1 == 1),1], color = "r",label = "Class 2")
plt.title("Trainning set", fontsize=15)
plt.legend(loc = 0)
plt.xlabel("X_1");
plt.ylabel("X_2");

np.random.seed(4)

X_2 = np.concatenate((X_class1, X_class2), axis = 0) #data_set
Y_2 = np.concatenate((np.zeros((N, 1)), 1*np.ones((N, 1))), axis = 0)

plt.figure(2)
plt.scatter(X_2[np.nonzero(Y_2 == 0),0], X_2[np.nonzero(Y_2 == 0),1], color = "b",label = "Class 1")
plt.scatter(X_2[np.nonzero(Y_2 == 1),0], X_2[np.nonzero(Y_2 == 1),1], color = "r",label = "Class 2")
plt.title("Test set", fontsize=15)
plt.legend(loc = 0)
plt.xlabel("X_1");
plt.ylabel("X_2");
```



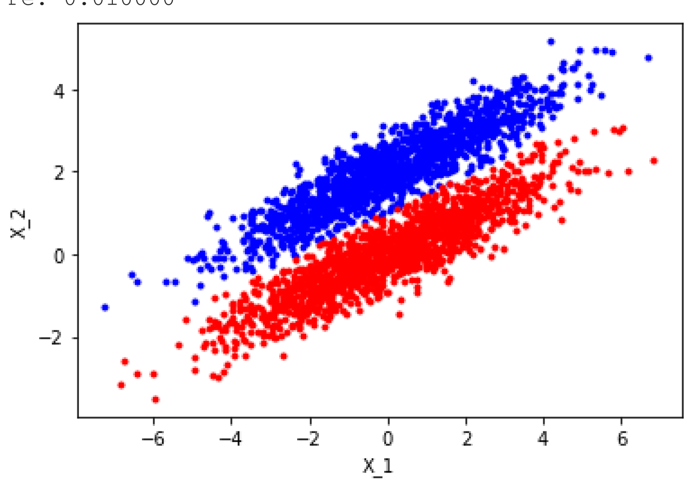


ii

31]:

```python
# Estimation of priori probabilities
# Classification using bayesian rule

P2 = P1 = 0.5
p1 = np.zeros(p)
p2 = np.zeros(p)



p1 = multivariate_normal_pdf_v2(X_2,m1 ,S);
p2 = multivariate_normal_pdf_v2(X_2,m2 ,S);
classes = np.zeros(p)

for i in range(0, p):
    if P1 * p1[i] > P2 * p2[i]:
        classes[i] = 0
    else:
        classes[i] = 1

Pe = 0
for i in range(0, p):
    if classes[i] != Y_2[i][0]:
        Pe += 1

Pe /= p
print('Pe: %f' % Pe)

plt.figure(1)
plt.plot(X_2[np.nonzero(classes == 0),0], X_2[np.nonzero(classes == 0),1], '.b')
plt.plot(X_2[np.nonzero(classes == 1),0], X_2[np.nonzero(classes == 1),1], '.r')

plt.xlabel("X_1");
plt.ylabel("X_2");
```

```
Pe: 0.010000
```



iii

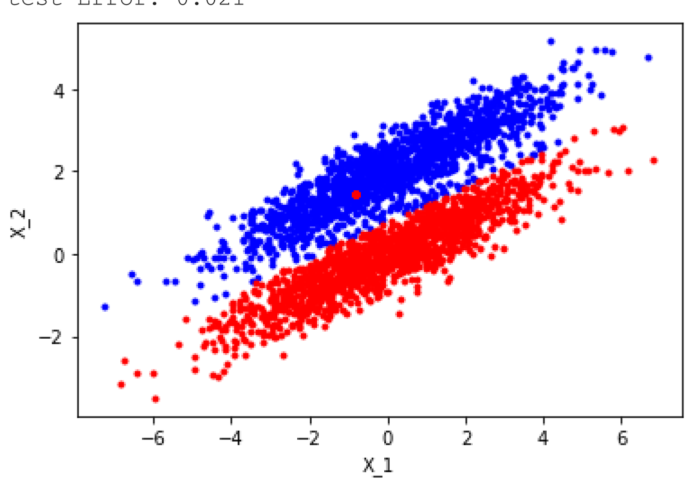
32]:

```python
def model(X_train, Y_train, X_test, Y_test, num_iterations , learning_rate , print_cost):

    # perform parameter initialization
    dim = X_train.shape[0]
    w = np.zeros((dim, 1))
    b = 0


    m = X_train.shape[1]

    costs = []

    # training
    for i in range(num_iterations):

        # calculate cost and gradients - forward propagation
        A = 1/(1+np.exp(-(np.dot(w.T,X_train)+b)))
        cost =  (-1/m)*(Y_train*np.log(A)+(1-Y_train)*np.log(1-A)).sum()

        # calculate gradients - backward propagation
        dw = np.dot(X_train,(A-Y_train).T)/m
        db = (A-Y_train).sum()/m


        # perform the update
        w = w - learning_rate*dw
        b = b - learning_rate*db


    A_test = 1/(1+np.exp(-(np.dot(w.T,X_test)+b)))
    Y_predict_test = np.around(A_test)

    # Print test Errors
    print("test Error: {} ".format( np.mean(np.abs(Y_predict_test - Y_test))))

    return Y_predict_test
```

33]:

```python
#Logistic regression

ypredict=model(X_1.T, Y_1.T, X_2.T, Y_2.T, num_iterations=20000 , learning_rate=0.001 , print_cost=True)

plt.figure(1)
plt.plot(X_2[np.nonzero(ypredict == 0),0], X_2[np.nonzero(ypredict == 0),1], '.b')
plt.plot(X_2[np.nonzero(ypredict == 1),0], X_2[np.nonzero(ypredict == 1),1], '.r')

plt.xlabel("X_1");
plt.ylabel("X_2");
```

```
test Error: 0.021
```



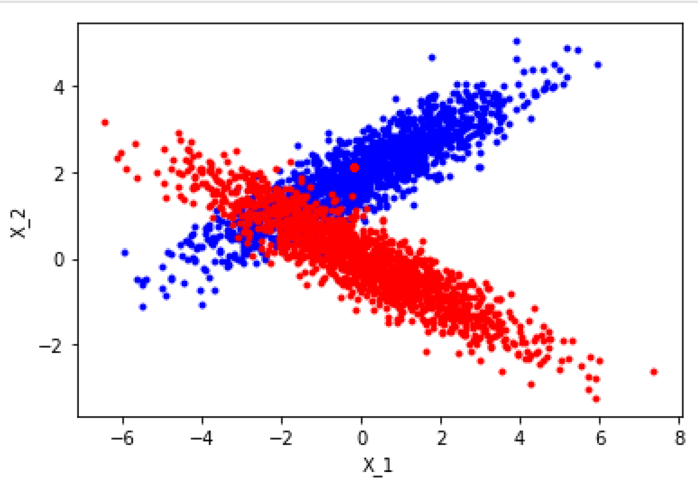
iv

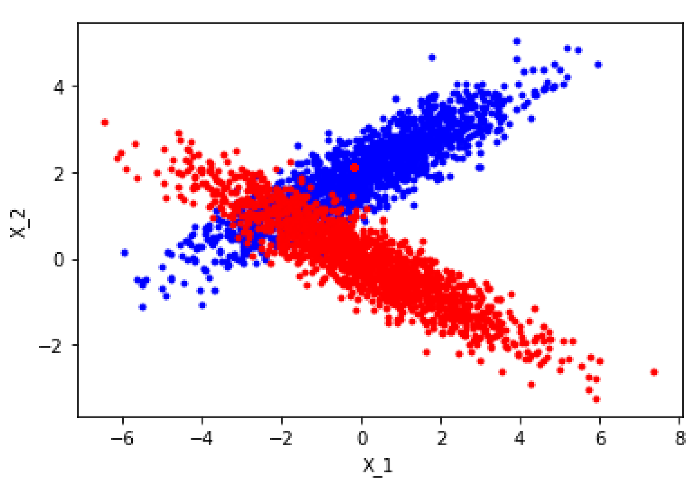
By using Bayesian classification with logistic regression, the probability error is 0.021333333333333333 this is bigger for logistic regression while comparing with Bayesian classification.

v

34]:

```python
#Different covariance

S1 = np.array([[4, 1.8], [1.8, 1]])
S2 = np.array([[4, -1.8], [-1.8, 1]])

X_class1 = np.random.multivariate_normal(m1,S1,N)
X_class2 = np.random.multivariate_normal(m2,S2,N)

X_1 = np.concatenate((X_class1, X_class2), axis = 0)
Y_1 = np.concatenate((0*np.ones((N, 1)), 1*np.ones((N, 1))), axis = 0)

plt.figure(1)
plt.plot(X_1[np.nonzero(Y_1 == 0),0], X_1[np.nonzero(Y_1 == 0),1], '.b')
plt.plot(X_1[np.nonzero(Y_1 == 1),0], X_1[np.nonzero(Y_1 == 1),1], '.r')

plt.xlabel("X_1");
plt.ylabel("X_2");

np.random.seed(5)

X_2 = np.concatenate((X_class1, X_class2), axis = 0)
Y_2 = np.concatenate((0*np.ones((N, 1)), 1*np.ones((N, 1))), axis = 0)

plt.figure(2)
plt.plot(X_2[np.nonzero(Y_2 == 0),0], X_2[np.nonzero(Y_2 == 0),1], '.b')
plt.plot(X_2[np.nonzero(Y_2 == 1),0], X_2[np.nonzero(Y_2 == 1),1], '.r')

plt.xlabel("X_1");
plt.ylabel("X_2");
```

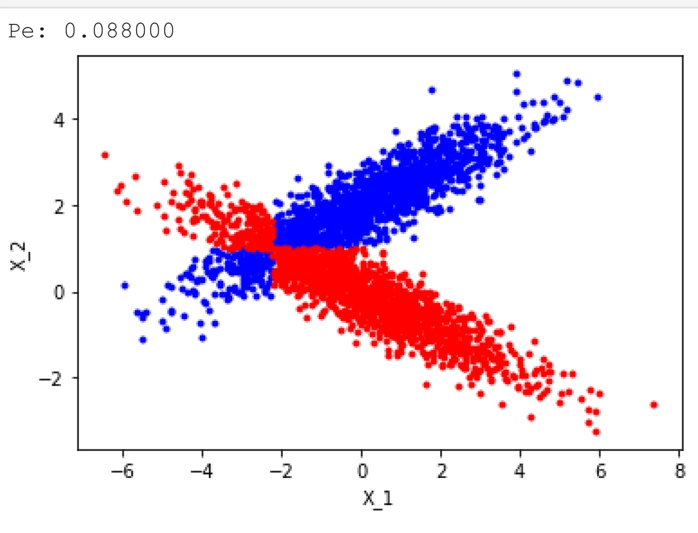




35]:

```python
P2 = P1 = 0.5
p1 = np.zeros(p)
p2 = np.zeros(p)


p1=multivariate_normal_pdf_v2(X_2,m1 ,S1);
p2=multivariate_normal_pdf_v2(X_2,m2 ,S2);
classes = np.zeros(p)

for i in range(0, p):
    if P1*p1[i] > P2*p2[i]:
        classes[i] = 0
    else:
        classes[i] = 1

Pe = 0
for i in range(0, p):
    if classes[i] != Y_2[i][0]:
        Pe += 1

Pe /= p
print('Pe: %f' % Pe)

plt.figure(1)
plt.plot(X_2[np.nonzero(classes == 0),0], X_2[np.nonzero(classes == 0),1], '.b')
plt.plot(X_2[np.nonzero(classes == 1),0], X_2[np.nonzero(classes == 1),1], '.r')

plt.xlabel("X_1");
plt.ylabel("X_2");
```

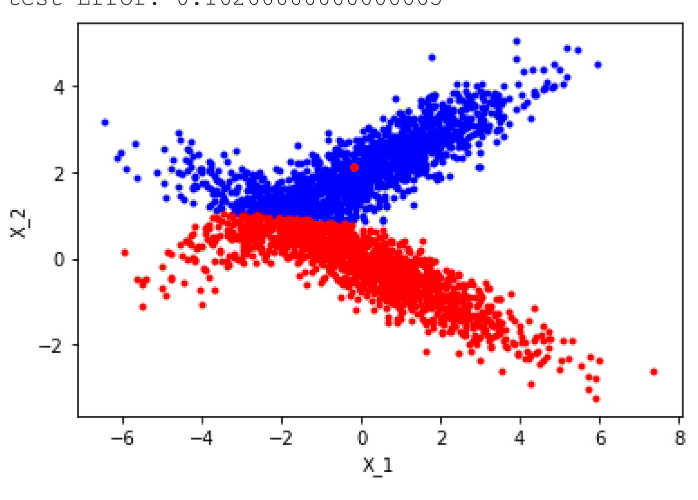
```
Pe: 0.088000
```



36]:

```python
ypredict=model(X_1.T, Y_1.T, X_2.T, Y_2.T, num_iterations=20000 , learning_rate=0.001 , print_cost=True)

plt.figure(1)
plt.plot(X_2[np.nonzero(ypredict == 0),0], X_2[np.nonzero(ypredict == 0),1], '.b')
plt.plot(X_2[np.nonzero(ypredict == 1),0], X_2[np.nonzero(ypredict == 1),1], '.r')

plt.xlabel("X_1");
plt.ylabel("X_2");
```

```
test Error: 0.16266666666666665
```



Following the repetition of previous steps, it is possible to observe that the error in logistic regression is higher while comparing it with Bayesian classification.

[ ]:

Question 5.2

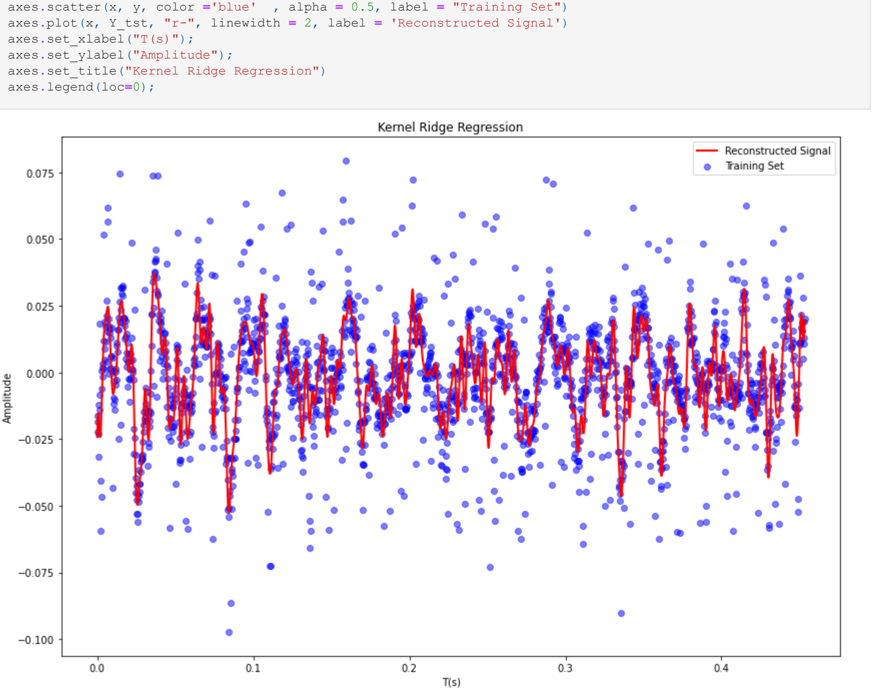
In [10]:
```python
import numpy as np
import math
import soundfile as sf
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
import matplotlib.pyplot as plt
import sys
import os
```

In [11]:
```python
def py_awgn(input_signal, snr_dB, rate=1.0):
    """ Addditive White Gaussian Noise (AWGN) Channel.

    Parameters
    __________
    input_signal : 1D ndarray of floats
        Input signal to the channel.

    snr_dB : float
        Output SNR required in dB.

    rate : float
        Rate of the a FEC code used if any, otherwise 1.

    Returns
    _______
    output_signal : 1D ndarray of floats
        Output signal from the channel with the specified SNR.
    """

    avg_energy = np.sum(np.dot(input_signal.conj().T, input_signal)) / input_signal.shape[0]
    snr_linear = 10 ** (snr_dB / 10.0)
    noise_variance = avg_energy / (2 * rate * snr_linear)

    if input_signal.dtype is np.complex:
        noise = np.array([np.sqrt(noise_variance) * np.random.randn(input_signal.shape[0]) * (1 + 1j)], ndmin=2
    else:
        noise = np.array([np.sqrt(2 * noise_variance) * np.random.randn(input_signal.shape[0])], ndmin=2)

    output_signal = input_signal + noise.conj().T

    return output_signal


def Kernel_Ridge(Xtrain, Ytrain, Xtest, sigma, lambda_):
    ' use of a kernel based solution to calculate the output Y_out'

    from scipy.spatial.distance import pdist, cdist, squareform

    # Design matrix K
    pairwise_sq_dists = squareform(pdist(Xtrain, 'sqeuclidean'))
    K  = np.exp(-pairwise_sq_dists / sigma**2)
    A  = K + lambda_ * np.identity(len(K))
    kx = np.exp(-cdist(Xtrain, Xtest,'sqeuclidean')/sigma**2)
    A  = np.linalg.inv(K + lambda_ * np.identity(len(K)))
    B  = np.matmul(kx.T,A)
    Y_out = np.matmul(B,Ytrain)

    return Y_out
```

In [12]:
```python
# Reading wav file. x corresponds to time instances (is., x_i in [0,1])
# fs is the sampling frequency
# Replace the name "BladeRunner.wav" with the name of the file
# you intend to use.

np.random.seed(6)

N = 2000
samples = 20000
ind = range(0, samples,int(samples/N))
strt = 150000
[data, fs] = sf.read('BladeRunner.wav')
sound = np.array(data[strt:(strt+samples+1), :], dtype=np.float32)
y = np.reshape(sound[ind, 0], newshape=(len(ind), 1))

Ts = 1/fs #sampling period

x = np.array(range(0, samples)).conj().transpose()*Ts
x = x[ind]
x = np.reshape(x, newshape=(x.shape[0], 1))


# Add white Gaussian noise
snr = 10   # dB
y = py_awgn(y, snr)

# add outliers
O = 0.8*np.max(np.abs(y))
percent = 0.1
M = int(math.floor(percent*N))
out_ind = np.random.choice(N, M, replace=False)
outs = np.sign(np.random.randn(M, 1))*O
y[out_ind] = y[out_ind] + outs
```

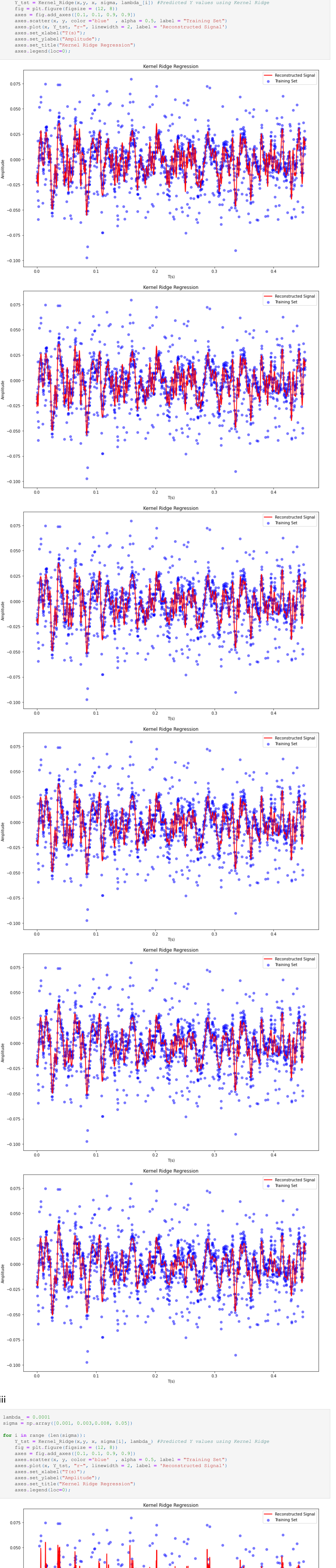
i

In [13]:
```python
lambda_ = 0.0001
sigma = 0.004

Y_tst = Kernel_Ridge(x,y, x, sigma, lambda_) #Predicted Y values using Kernel Ridge

fig = plt.figure(figsize = (12, 8))
axes = fig.add_axes([0.1, 0.1, 0.9, 0.9])
axes.scatter(x, y, color ='blue'  , alpha = 0.5, label = "Training Set")
axes.plot(x, Y_tst, "r-", linewidth = 2, label = 'Reconstructed Signal')
axes.set_xlabel("T(s)");
axes.set_ylabel("Amplitude");
axes.set_title("Kernel Ridge Regression")
axes.legend(loc=0);
```



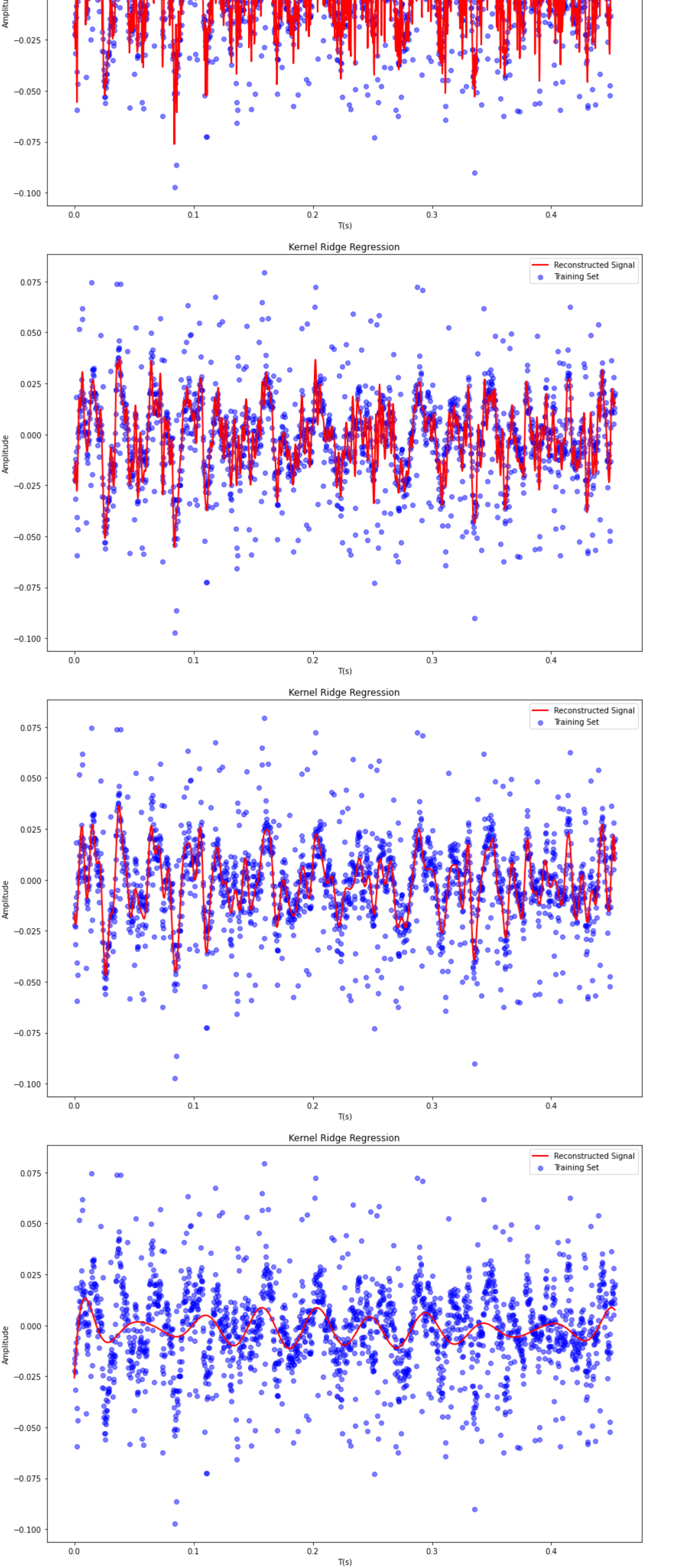
ii

In [14]:
```python
sigma = 0.004
lambda_ = np.array([10**-6, 10**-5,0.0005, 0.001,0.01, 0.05])

for i in range (len(lambda_)):
    Y_tst = Kernel_Ridge(x,y, x, sigma, lambda_[i]) #Predicted Y values using Kernel Ridge
    fig = plt.figure(figsize = (12, 8))
    axes = fig.add_axes([0.1, 0.1, 0.9, 0.9])
    axes.scatter(x, y, color ='blue'  , alpha = 0.5, label = "Training Set")
    axes.plot(x, Y_tst, "r-", linewidth = 2, label = 'Reconstructed Signal')
    axes.set_xlabel("T(s)");
    axes.set_ylabel("Amplitude");
    axes.set_title("Kernel Ridge Regression")
    axes.legend(loc=0);
```



iii

In [15]:
```python
lambda_ = 0.0001
sigma = np.array([0.001, 0.003,0.008, 0.05])

for i in range (len(sigma)):
    Y_tst = Kernel_Ridge(x,y, x, sigma[i], lambda_) #Predicted Y values using Kernel Ridge
    fig = plt.figure(figsize = (12, 8))
    axes = fig.add_axes([0.1, 0.1, 0.9, 0.9])
    axes.scatter(x, y, color ='blue'  , alpha = 0.5, label = "Training Set")
    axes.plot(x, Y_tst, "r-", linewidth = 2, label = 'Reconstructed Signal')
    axes.set_xlabel("T(s)");
    axes.set_ylabel("Amplitude");
    axes.set_title("Kernel Ridge Regression")
    axes.legend(loc=0);
```



iv

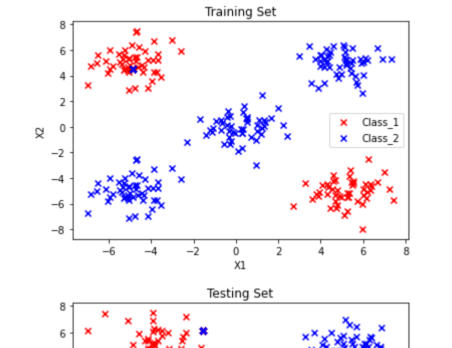
By observing the graphs it cand be said that when increasing the value of lambda, the quality is reduced by sigma due to loss of the outliers. The effect of sigma is greater than lam Effect of lambda is not as much as sigma, since sigma essential adds more noise variance

Question 5.3

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

def nn_model(X, Y, n_h, num_iterations , learning_rate , print_cost ):

    np.random.seed(3)

    ## NN definition
    n_x = X.shape[0]    # size of input layer
    n_h = n_h
    n_y = Y.shape[0]   # size of output layer


    # parameter initialization
    W1 = np.random.randn(n_h, n_x)*0.1
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h)*0.1
    b2 = np.zeros((n_y, 1))

    # gradient descent loop
    for i in range(0, num_iterations):

        # Forward propagation
        A1 = np.dot(W1, X) + b1
        Z1 = np.tanh(A1)
        A2 = np.dot(W2, Z1) + b2
        Z2 = A2

        # compute the cost
        m = Y.shape[1]
        cost = np.sum((Y-Z2)**2)/m

        # perform back-propagation
        dA2 = Z2 - Y
        dW2 = np.matmul(dA2, Z1.T)/m
        db2 = np.sum(dA2, axis = 1, keepdims = True)/m
        dA1 = np.multiply(np.matmul(W2.T, dA2), (1-np.power(Z1, 2)))
        dW1 = np.matmul(dA1, X.T)/m
        db1 = np.sum(dA1, axis = 1, keepdims = True)/m

        # Parameter update
        W1 = W1-learning_rate*dW1
        b1 = b1-learning_rate*db1
        W2 = W2-learning_rate*dW2
        b2 = b2-learning_rate*db2

              # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}

    return parameters

def nn_predict(parameters, X):

    """
    Arguments:
    parameters -- python dictionary containing trained parameters
    X -- input data of size (n_x, m)

    Returns:
    predictions -- vector of predictions of our model (red: 0 / blue: 1)
    """

    # unpack the parameters
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Forward propagation
    A1 = np.dot(W1,X)  + b1
    Z1 = np.tanh(A1)
    A2 = np.dot(W2,Z1) + b2
    Z2 = A2      #  output latyer's sigmoid transfer function

    predictions = np.around(Z2)

    return predictions

def plot_decision_boundary(model, X, y):

    # set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01

    # generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # predict the function value of the whole grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    # plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap = plt.cm.Spectral)
    plt.ylabel("x2")
    plt.xlabel("x1")
    plt.scatter(X[0, :], X[1, :], c = y, cmap = plt.cm.Spectral)

def mixt_model(m, S, N, sed):
    """
    m :  matrix for means of the subclasses : (l,c) , where c is number of classes, l dimensions of input space
    S :  matrix for covariances of the subclasses : (l, l, c)
    N :  array with the number of elements per class : (l, c)
    """

    np.random.seed(sed)
    l = m.shape[0]         # dimension of the space
    c = m.shape[1]         # number of gaussian sub-classes
    Ntotal = np.sum(N)
    X = []

    for i in range(0,c):
        Xc = np.random.multivariate_normal(np.array(m[:, i]).T, np.array(S[:, :, i]), N[i]).T
        X.append(Xc)
    X = np.hstack(X)

    return X[:, np.random.permutation(Ntotal)]
```

i

In [2]:
```python
seed = 10
D = 2  # Dimension
m1 = np.array([[-5, 5], [5, -5]]).T  #mean for class one
m2 = np.array([[-5, -5], [0, 0], [5, 5]]).T #mean for class two
c1 = m1.shape[1]  # number of gaussians per class 1
c2 = m2.shape[1]  # number of gaussians per class 2

#Training Set Generation
N1 = np.array([50,50])
S1 = np.zeros(shape=(D,D, c1))
s_1=1 #Variance
for i in range(0, c1):
    S1[:, :, i] = np.array(s_1*np.eye(2))

w1trainning = mixt_model(m1, S1, N1, seed) #training set for first class
N2 = np.array([50,50,50])
S2 = np.zeros(shape=(D, D, c2))
for i in range(0, c2):
    S2[:, :, i] = s_1 * np.eye(D)
w2trainning= mixt_model(m2, S2, N2, seed) #training set for second class
Xtrain = np.concatenate((w1trainning,  w2trainning), axis=1);
Ytrain = np.concatenate((np.zeros(shape=(1, np.sum(N1))), np.ones(shape=(1, np.sum(N2)))), axis=1)

#Test Set Generation
seed = 8
w1test = mixt_model(m1, S1, N1, seed)#test set for first class
w2test= mixt_model(m2, S2, N2, seed) #test set for second class
Xtest= np.concatenate((w1test,  w2test), axis=1);
Ytest=np.concatenate((np.zeros(shape=(1, np.sum(N1))), np.ones(shape=(1, np.sum(N2)))), axis=1)

plt.figure(1)
plt.scatter(Xtrain[0,np.nonzero(Ytrain == 0)], Xtrain[1,np.nonzero(Ytrain ==0)]
            , marker = "x", color = "r", label = "Class_1");
plt.scatter(Xtrain[0,np.nonzero(Ytrain == 1)], Xtrain[1,np.nonzero(Ytrain ==1)]
            , marker = "x", color = "b", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Training Set");
plt.figure(2)
plt.scatter(Xtest[0,np.nonzero(Ytrain == 0)], Xtest[1,np.nonzero(Ytrain ==0)]
            , marker = "x", color = "r", label = "Class_1");
plt.scatter(Xtest[0,np.nonzero(Ytrain == 1)], Xtest[1,np.nonzero(Ytrain ==1)]
            , marker = "x", color = "b", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Testing Set");
```



ii

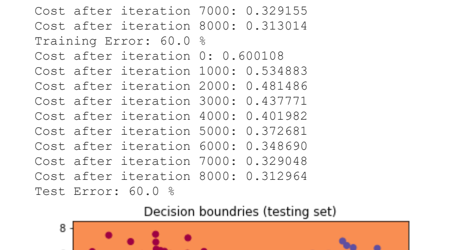
```python
parameters_train = nn_model(Xtrain,Ytrain,n_h = 2,num_iterations = 9000,learning_rate = 0.01,print_cost = True [illegible]

#Compute Training and Test Errors
Y_predict_train = nn_predict(parameters_train, Xtrain)
print("Training Error: {} %".format(np.mean(np.abs(Y_predict_train-Ytrain))*100))

parameters_test = nn_model(Xtest,Ytest,n_h = 2,num_iterations = 9000,learning_rate = 0.01,print_cost = True [illegible]
Y_predict_test = nn_predict(parameters_test, Xtest)
print("Test Error: {} %".format(np.mean(np.abs(Y_predict_test-Ytest))*100))

#Plotting Decision Boundaries generated from the network
plot_decision_boundary(lambda x:nn_predict(parameters_test,x.T),Xtest,Ytest.ravel())
plt.title("Decision boundries (testing set)", fontsize=15);
```

```
Cost after iteration 0: [illegible]
Cost after iteration 1000: [illegible]
Cost after iteration 2000: [illegible]
Cost after iteration 3000: [illegible]
Cost after iteration 4000: [illegible]
Cost after iteration 5000: [illegible]
Cost after iteration 6000: [illegible]
Cost after iteration 7000: [illegible]
Cost after iteration 8000: [illegible]
Training Error: 40.0 %
Cost after iteration 0: [illegible]
Cost after iteration 1000: [illegible]
Cost after iteration 2000: [illegible]
Cost after iteration 3000: [illegible]
Cost after iteration 4000: [illegible]
Cost after iteration 5000: [illegible]
Cost after iteration 6000: [illegible]
Cost after iteration 7000: [illegible]
Cost after iteration 8000: [illegible]
Test Error: 40.0 %
```

iii

```python
parameters_train = nn_model(Xtrain,Ytrain,n_h = 2,num_iterations = 9000,learning_rate = 0.0001,print_cost = Tru [illegible]

#Compute Training and Test Errors
Y_predict_train = nn_predict(parameters_train, Xtrain)
print("Training Error: {} %".format(np.mean(np.abs(Y_predict_train-Ytrain))*100))

parameters_test = nn_model(Xtest,Ytest,n_h = 2,num_iterations = 9000,learning_rate = 0.0001,print_cost = True [illegible]
Y_predict_test = nn_predict(parameters_test, Xtest)
print("Test Error: {} %".format(np.mean(np.abs(Y_predict_test-Ytest))*100))

#Plotting Decision Boundaries generated from the network
plot_decision_boundary(lambda x:nn_predict(parameters_test,x.T),Xtest,Ytest.ravel())
plt.title("Decision boundries (testing set)", fontsize=15);
```

```
Cost after iteration 0: [illegible]
Cost after iteration 1000: [illegible]
Cost after iteration 2000: [illegible]
Cost after iteration 3000: [illegible]
Cost after iteration 4000: [illegible]
Cost after iteration 5000: [illegible]
Cost after iteration 6000: [illegible]
Cost after iteration 7000: [illegible]
Cost after iteration 8000: [illegible]
Training Error: 40.0 %
Cost after iteration 0: [illegible]
Cost after iteration 1000: [illegible]
Cost after iteration 2000: [illegible]
Cost after iteration 3000: [illegible]
Cost after iteration 4000: [illegible]
Cost after iteration 5000: [illegible]
Cost after iteration 6000: [illegible]
Cost after iteration 7000: [illegible]
Cost after iteration 8000: [illegible]
Test Error: 40.0 %
```

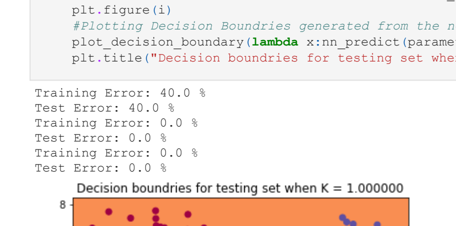


As evident grom the graph above, with the parameters used hh time, the model is not classifying, all of the data points are place with the same background color meaning that it did not classify.

iv

```python
n_h = np.array([1, 4, 20])
for i in range (len(n_h)):
    parameters_train = nn_model(Xtrain,Ytrain,n_h = n_h[i],num_iterations = 9000,learning_rate = 0.01,print_cost = Fa [illegible]

    #Compute Training and Test Errors
    Y_predict_train = nn_predict(parameters_train, Xtrain)
    print("Training Error: {} %".format(np.mean(np.abs(Y_predict_train-Ytrain))*100))

    parameters_test = nn_model(Xtest,Ytest,n_h = n_h[i],num_iterations = 9000,learning_rate = 0.01,print_cost = False [illegible]
    Y_predict_test = nn_predict(parameters_test, Xtest)
    print("Test Error: {} %".format(np.mean(np.abs(Y_predict_test-Ytest))*100))
    plt.figure(i)
    #Plotting Decision Boundaries generated from the network
    plot_decision_boundary(lambda x:nn_predict(parameters_test,x.T),Xtest,Ytest.ravel())
    plt.title("Decision boundries for testing set when K = %f" %n_h[i], fontsize=10);
```

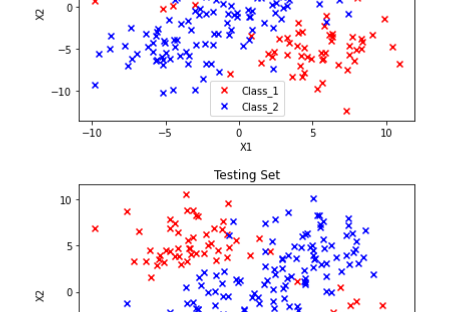
```
Training Error: [illegible] %
Test Error: [illegible] %
Training Error: [illegible] %
Test Error: [illegible] %
Training Error: [illegible] %
Test Error: [illegible] %
```



It can be seen from the graphs that when the learning rate is low, the model can not classify the data pointsbecause it is harder to reach a conclusion of what class the points belong to.

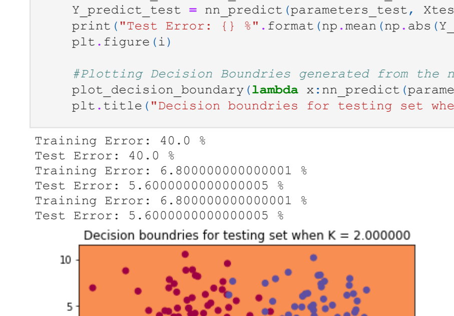
v

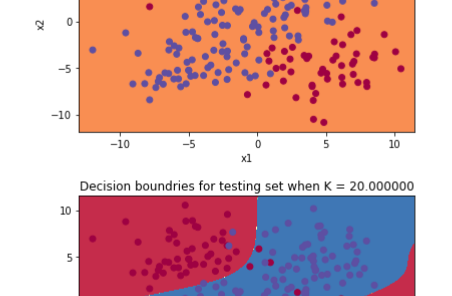
```python
seed =10
D = 2  # Dimension
m1 = np.array([[-5, 5], [5, -5]]).T #mean for class one
m2 = np.array([[-5, -5], [0, 0], [5, 5]]).T #mean for class two
c1 = m1.shape[1]  # number of gaussians per class 1
c2 = m2.shape[1]  # number of gaussians per class 2

#Training Set Generation
N1 = np.array([50,50])
S1 = np.zeros(shape=(D,D, c1))
s_1=5 #Variance
for i in range(0, c1):
    S1[:, :, i] = np.array(s_1*np.eye(2))

w1trainning = mixt_model(m1, S1, N1, seed) #training set for first class
N2 = np.array([50,50,50])
S2 = np.zeros(shape=(D, D, c2))
for i in range(0, c2):
    S2[:, :, i] = s_1 * np.eye(D)
w2trainning= mixt_model(m2, S2, N2, seed) #training set for second class
Xtrain = np.concatenate((w1trainning,  w2trainning), axis=1);
Ytrain = np.concatenate((np.zeros(shape=(1, np.sum(N1))), np.ones(shape=(1, np.sum(N2)))), axis=1)

#Test Set Generation
seed = 8
w1test = mixt_model(m1, S1, N1, seed)#test set for first class
w2test= mixt_model(m2, S2, N2, seed) #test set for second class
Xtest= np.concatenate((w1test,  w2test), axis=1);
Ytest=np.concatenate((np.zeros(shape=(1, np.sum(N1))), np.ones(shape=(1, np.sum(N2)))), axis=1)

plt.figure(1)
plt.scatter(Xtrain[0,np.nonzero(Ytrain == 0)], Xtrain[1,np.nonzero(Ytrain ==0)]
            , marker = "x", color = "r", label = "Class_1");
plt.scatter(Xtrain[0,np.nonzero(Ytrain == 1)], Xtrain[1,np.nonzero(Ytrain ==1)]
            , marker = "x", color = "b", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Training Set");
plt.figure(2)
plt.scatter(Xtest[0,np.nonzero(Ytrain == 0)], Xtest[1,np.nonzero(Ytrain ==0)]
            , marker = "x", color = "r", label = "Class_1");
plt.scatter(Xtest[0,np.nonzero(Ytrain == 1)], Xtest[1,np.nonzero(Ytrain ==1)]
            , marker = "x", color = "b", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Testing Set");
```

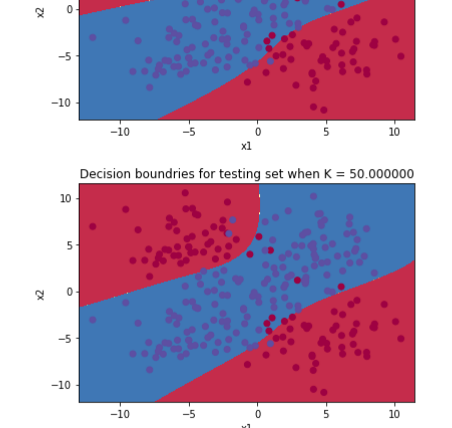



```python
n_h = np.array([2, 20, 50])
for i in range (len(n_h)):
    parameters_train = nn_model(Xtrain,Ytrain,n_h = n_h[i],num_iterations = 9000,learning_rate = 0.01,print_cost = Fa [illegible]

    #Compute Training and Test Errors
    Y_predict_train = nn_predict(parameters_train, Xtrain)
    print("Training Error: {} %".format(np.mean(np.abs(Y_predict_train-Ytrain))*100))

    parameters_test = nn_model(Xtest,Ytest,n_h = n_h[i],num_iterations = 9000,learning_rate = 0.01,print_cost = False [illegible]
    Y_predict_test = nn_predict(parameters_test, Xtest)
    print("Test Error: {} %".format(np.mean(np.abs(Y_predict_test-Ytest))*100))
    plt.figure(i)
    #Plotting Decision Boundaries generated from the network
    plot_decision_boundary(lambda x:nn_predict(parameters_test,x.T),Xtest,Ytest.ravel())
    plt.title("Decision boundries for testing set when K = %f" %n_h[i], fontsize=10);
```

```
Training Error: [illegible] %
Test Error: [illegible] %
Training Error: [illegible] %
Test Error: [illegible] %
Training Error: [illegible] %
Test Error: [illegible] %
```








```python
n_h = np.array([2, 20, 50])
for i in range (len(n_h)):
    parameters_train = nn_model(Xtrain,Ytrain,n_h = n_h[i],num_iterations = 9000,learning_rate = 0.001,print_cost = F [illegible]

    #Compute Training and Test Errors
    Y_predict_train = nn_predict(parameters_train, Xtrain)
    print("Training Error: {} %".format(np.mean(np.abs(Y_predict_train-Ytrain))*100))

    parameters_test = nn_model(Xtest,Ytest,n_h = n_h[i],num_iterations = 9000,learning_rate = 0.001,print_cost = False [illegible]
    Y_predict_test = nn_predict(parameters_test, Xtest)
    print("Test Error: {} %".format(np.mean(np.abs(Y_predict_test-Ytest))*100))
    plt.figure(i)
    #Plotting Decision Boundaries generated from the network
    plot_decision_boundary(lambda x:nn_predict(parameters_test,x.T),Xtest,Ytest.ravel())
    plt.title("Decision boundries for testing set when K = %f" %n_h[i], fontsize=10);
```

```
Training Error: [illegible] %
Test Error: [illegible] %
Training Error: [illegible] %
Test Error: [illegible] %
Training Error: 19.6 %
Test Error: [illegible] %
```