Assignment E: 5.1

In [ ]:

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

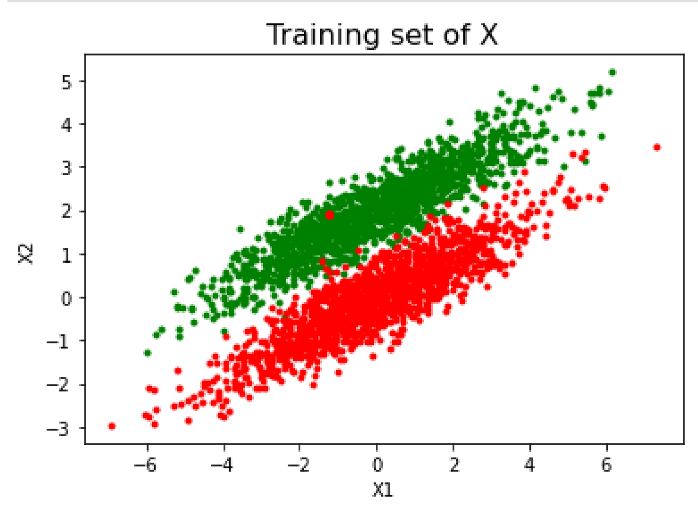
In [20]:

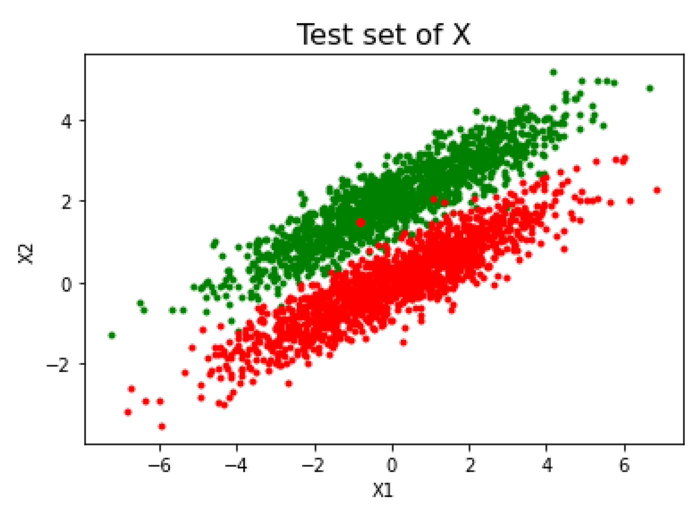
```
def multivariate_normal_pdf(x, mean, sigma):
    l = x.shape[0]
    det_S = np.linalg.det(sigma)
    norm_const = 1.0/((2.0*np.pi)**(l/2.0)*np.sqrt(det_S))
    inv_S = np.linalg.inv(sigma)
    a1 = np.dot(np.dot((x-mean), inv_S), (x-mean))

    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)
```

Part i: Generating and Plotting the data sets

In [21]:

```
myu_1 = np.array([0, 2]) #First class of mean
myu_2 = np.array([0, 0]) #Second class of mean
S = np.array([[4, 1.8], [1.8, 1]]) #covariance matrix of the given class of mean

N = 1500 # Per class, the number of data points
p = 2*N

#(i) Forming and plotting data set
first_section_of_X1 = np.random.multivariate_normal(myu_1,S,N)
second_section_of_X1 = np.random.multivariate_normal(myu_2, S,N)

X1 = np.concatenate((first_section_of_X1,second_section_of_X1), axis = 0) #data_set
Y1 = np.concatenate((0*np.ones((N, 1)),1*np.ones((N, 1))), axis = 0)

plt.figure(1)
plt.plot(X1[np.nonzero(Y1 == 0),0], X1[np.nonzero(Y1 == 0),1], '.g')
plt.plot(X1[np.nonzero(Y1 == 1),0], X1[np.nonzero(Y1 == 1),1], '.r')
plt.title(r'Training set of X', fontsize=16)

plt.xlabel("X1");
plt.ylabel("X2");

#(i) Forming and plotting data set
np.random.seed(5)
first_section_of_X2 = np.random.multivariate_normal(myu_1,S,N)
second_section_of_X2 = np.random.multivariate_normal(myu_2, S,N)
X2 = np.concatenate((first_section_of_X2,second_section_of_X2), axis = 0) #data_set
Y2 = np.concatenate((0*np.ones((N, 1)),1*np.ones((N, 1))), axis = 0)

plt.figure(2)
plt.plot(X2[np.nonzero(Y2 == 0),0], X2[np.nonzero(Y2 == 0),1], '.g')
plt.plot(X2[np.nonzero(Y2 == 1),0], X2[np.nonzero(Y2 == 1),1], '.r')
plt.title(r'Test set of X', fontsize=16)

plt.xlabel("X1");
plt.ylabel("X2");
```



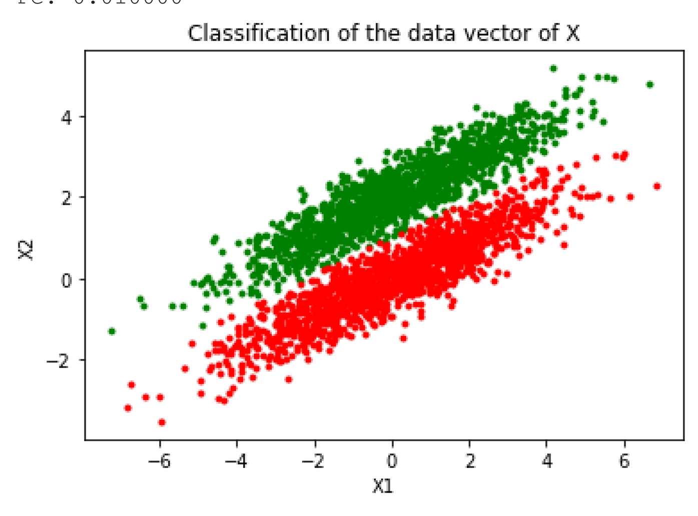


Part ii: Clssifying the data vector of X

In [22]:

```
P1 = 0.5
P2 = P1
p1 = np.zeros(p)
p2 = np.zeros(p)
# For each data point, a pdf estimate is made.

p1=multivariate_normal_pdf_v2(X2,myu_1 ,S); # Prior_propability * Gaussian_PDF
p2=multivariate_normal_pdf_v2(X2,myu_2 ,S);
classes = np.zeros(p)
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  # Probability error
for i in range(0, p):
    if classes[i] != Y2[i][0]:
        Pe += 1

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

plt.figure(1)
plt.plot(X2[np.nonzero(classes == 0),0], X2[np.nonzero(classes == 0),1], '.g')
plt.plot(X2[np.nonzero(classes == 1),0], X2[np.nonzero(classes == 1),1], '.r')
plt.title(r'Classification of the data vector of X', fontsize=12)

plt.xlabel("X1");
plt.ylabel("X2");
```

```
Pe: 0.010000
```



Part iii: Performing logistic regression

In [23]:

```
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 Level
    for i in range(num_iterations):

        # cost and gradients calculation (forward propagation)
        A = 1/(1+np.exp(-(np.dot(w.T,X_train)+b))) # determine the activation: Dimensions: (1, number of exampl
        cost =  (-1/m)*(Y_train*np.log(A)+(1-Y_train)*np.log(1-A)).sum() # Assessing cost

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


        # carry out the upgrade
        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
```

In [24]:

```
y_predict=model(X1.T, Y1.T, X2.T, Y2.T, num_iterations=20000 , learning_rate=0.001 , print_cost=True)

plt.figure(1)
plt.plot(X2[np.nonzero(y_predict == 0),0], X2[np.nonzero(y_predict == 0),1], '.g')
plt.plot(X2[np.nonzero(y_predict == 1),0], X2[np.nonzero(y_predict == 1),1], '.r')
plt.title(r'Classification of error', fontsize=12)

plt.xlabel("X1");
plt.ylabel("X2");
```

```
test Error: 0.021333333333333333
```

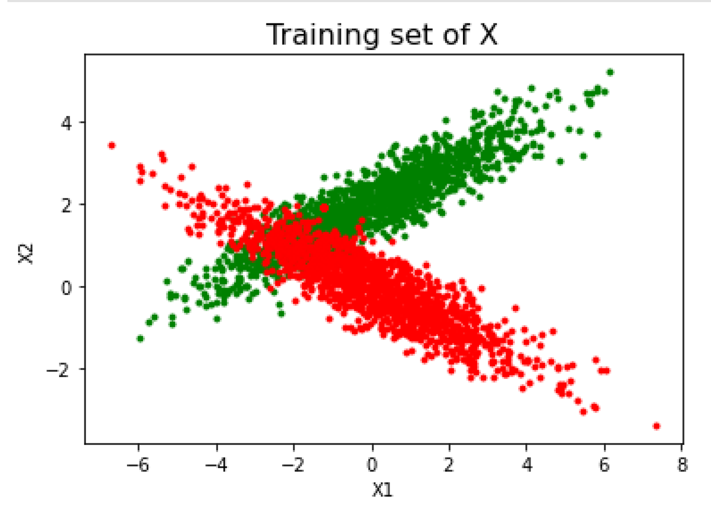
Part iv: Comments

By performing Bayesian classification and logistic regression, we can see that the probability error is 0.021333333333333333 which is higher for logistic regression compared Bayesian classification.

Part v: Repeating the step from i-iv

In [25]:

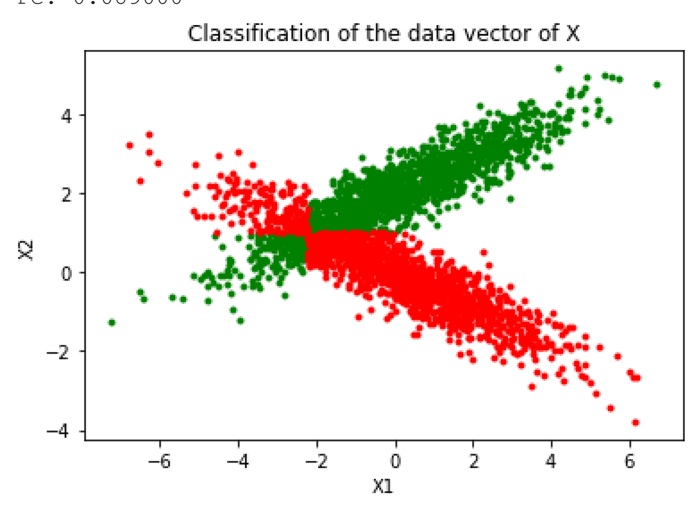
```
S1 = np.array([[4, 1.8], [1.8, 1]])
S2 = np.array([[4, -1.8], [-1.8, 1]]) #Given covariance matrix

first_section_of_X1 = np.random.multivariate_normal(myu_1,S1,N)
second_section_of_X1 = np.random.multivariate_normal(myu_2, S2,N)
X1 = np.concatenate((first_section_of_X1,second_section_of_X1), axis = 0) #data_set
Y1 = np.concatenate((0*np.ones((N, 1)), 1*np.ones((N, 1))), axis = 0)

plt.figure(1)
plt.plot(X1[np.nonzero(Y1 == 0),0], X1[np.nonzero(Y1 == 0),1], '.g')
plt.plot(X1[np.nonzero(Y1 == 1),0], X1[np.nonzero(Y1 == 1),1], '.r')

plt.xlabel("X1");
plt.ylabel("X2");

#(i) Forming and plotting data set
np.random.seed(5)
first_section_of_X2 = np.random.multivariate_normal(myu_1,S1,N)
second_section_of_X2 = np.random.multivariate_normal(myu_2, S2,N)
X2 = np.concatenate((first_section_of_X2,second_section_of_X2), axis = 0) #data_set
Y2 = np.concatenate((0*np.ones((N, 1)), 1*np.ones((N, 1))), axis = 0)
plt.title(r'Training set of X', fontsize=16)

plt.figure(2)
plt.plot(X2[np.nonzero(Y2 == 0),0], X2[np.nonzero(Y2 == 0),1], '.g')
plt.plot(X2[np.nonzero(Y2 == 1),0], X2[np.nonzero(Y2 == 1),1], '.r')
plt.title(r'Test set of X', fontsize=16)

plt.xlabel("X1");
plt.ylabel("X2");
```



In [26]:

```
P1 = P2 = 0.5
p1 = np.zeros(p)
p2 = np.zeros(p)
# For each data point, a pdf estimate is made.

p1=multivariate_normal_pdf_v2(X2,myu_1 ,S1); # Prior propability * Gaussian PDF
p2=multivariate_normal_pdf_v2(X2,myu_2 ,S2);
classes = np.zeros(p)
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  # Probability of error
for i in range(0, p):
    if classes[i] != Y2[i][0]:
        Pe += 1

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

plt.figure(1)
plt.plot(X2[np.nonzero(classes == 0),0], X2[np.nonzero(classes == 0),1], '.g')
plt.plot(X2[np.nonzero(classes == 1),0], X2[np.nonzero(classes == 1),1], '.r')
plt.title(r'Classification of the data vector of X', fontsize=12)

plt.xlabel("X1");
plt.ylabel("X2");
```

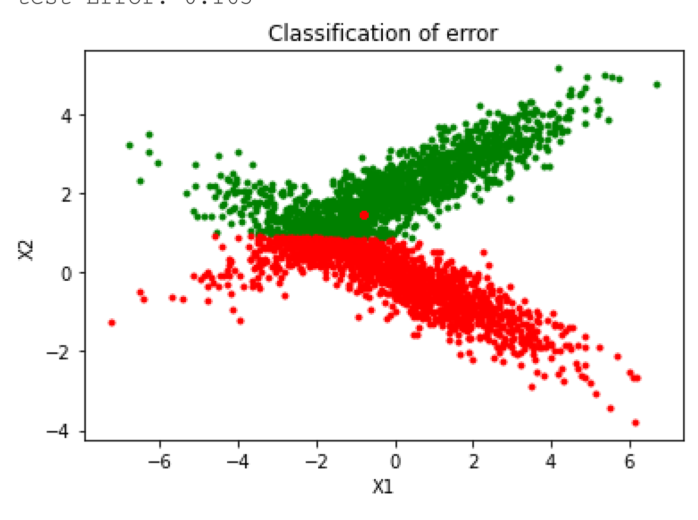
```
Pe: 0.089000
```



In [27]:

```
ypredict=model(X1.T, Y1.T, X2.T, Y2.T, num_iterations=20000 , learning_rate=0.001 , print_cost=True)

plt.figure(1)
plt.plot(X2[np.nonzero(ypredict == 0),0], X2[np.nonzero(ypredict == 0),1], '.g')
plt.plot(X2[np.nonzero(ypredict == 1),0], X2[np.nonzero(ypredict == 1),1], '.r')
plt.title(r'Classification of error', fontsize=12)

plt.xlabel("X1");
plt.ylabel("X2");
```

```
test Error: 0.163
```



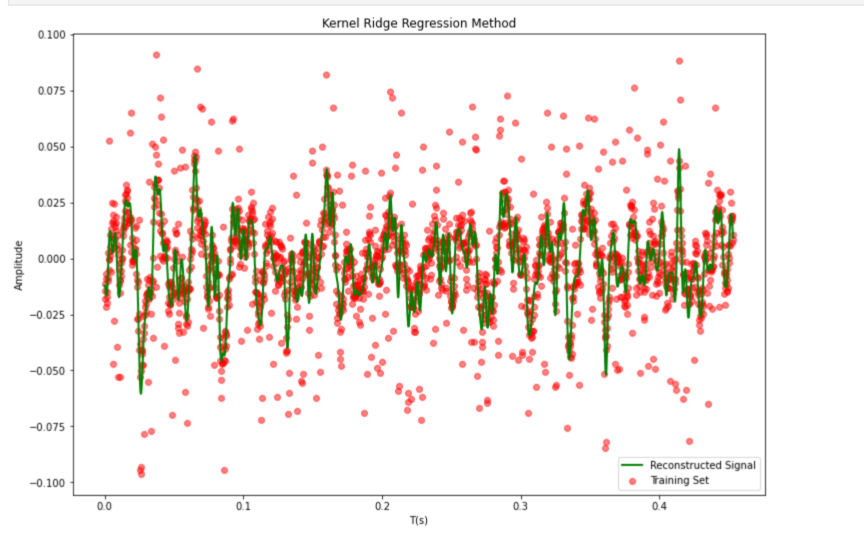
After repeating step i-iv, we can see that error is quite higher in in logistic regression which is 0.163 compared to Bayesian classification

Assignment E: 5.2

In [ ]:
```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


def py_awgn(input_signal, SNR_dB, rate=1.0):
    """ Adddditive White Gaussian Noise (AWGN) Channel.

    Parameters
    __________
    input_signal : 1D ndarray of floaT_s
        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 floaT_s
        Output signal from the channel with the specified SNR.
    """

    average_energy = np.sum(np.dot(input_signal.conj().T, input_signal)) / input_signal.shape[0]
    SNR_linear = 10 ** (SNR_dB / 10.0)
    noise_variance = average_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(X_training, Y_training, X_testing, SIGMA, LAMDA):
    ' use of a kernel based solution to calculate the output Y_out'

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

    # Design matrix K
    pairwise_sqeuclidean_disT_s = squareform(pdist(X_training, 'sqeuclidean'))
    K  = np.exp(-pairwise_sqeuclidean_disT_s / SIGMA**2)
    A  = K + LAMDA * np.identity(len(K))
    k_x = np.exp(-cdist(X_training, X_testing,'sqeuclidean')/SIGMA**2)
    A  = np.linalg.inv(K + LAMDA * np.identity(len(K)))
    B  = np.matmul(k_x.T,A)
    Y_out = np.matmul(B,Y_training)

    return Y_out


np.random.seed(0)
# 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.

N = 2000
samp = 20000
indc = range(0, samp,int(samp/N))
start = 150000
[data, fs] = sf.read('BladeRunner.wav')
sound = np.array(data[start:(start+samp+1), :], dtype=np.float32)
y = np.reshape(sound[indc, 0], newshape=(len(indc), 1))
T_s = 1/fs  # h periodos deigmatolipsias
x = np.array(range(0, samp)).conj().transpose()*T_s  # oi xronikes stigmes tis deigmatolipsias
x = x[indc]
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)
ouT_s = np.sign(np.random.randn(M, 1))*O
y[out_ind] = y[out_ind] + ouT_s
```

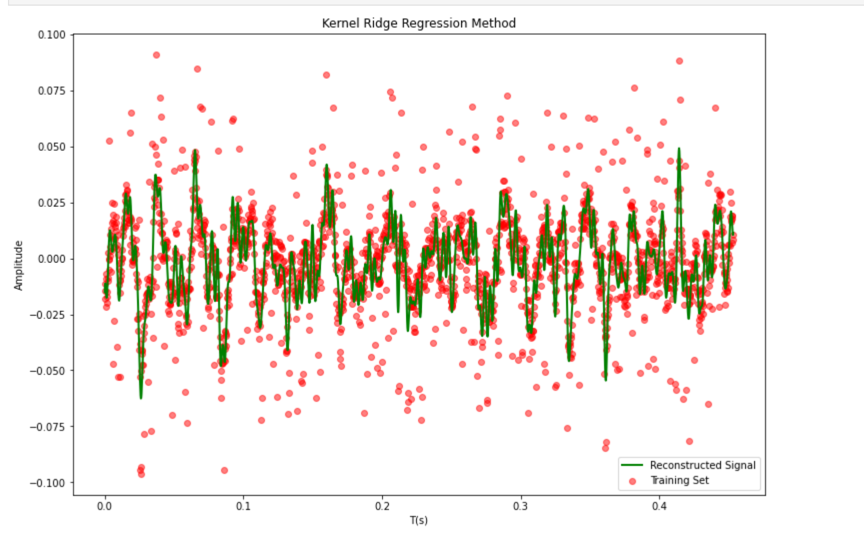
Part i: Performing reconstructed data samples using the kernel bridge regression

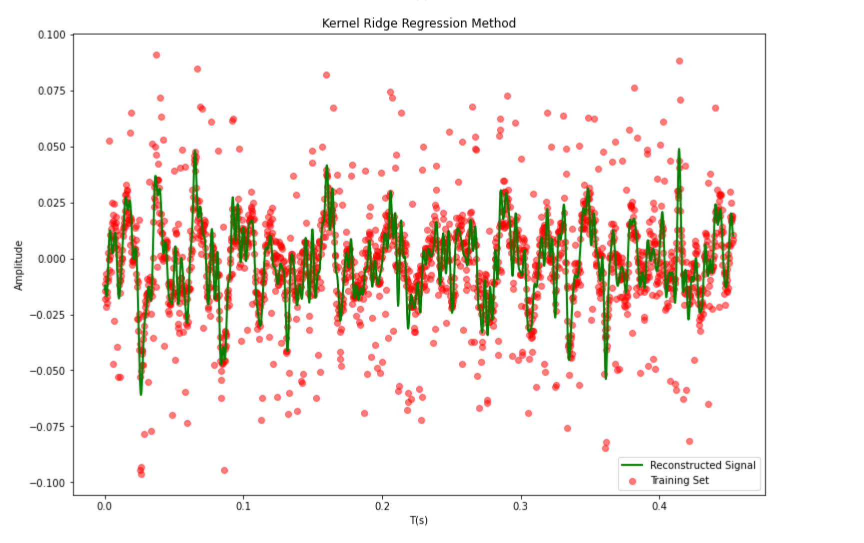
In [2]:
```python
SIGMA = 0.004
LAMDA = 0.0001

Y_testing = Kernel_Ridge(x,y, x, SIGMA, LAMDA) # Kernel Ridge predicted Y values.

fig = plt.figure(figsize = (12, 8))
axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])
axes.scatter(x, y, color ='red'  , alpha = 0.5, label = "Training Set")
axes.plot(x, Y_testing, "g-", linewidth = 2, label = 'Reconstructed Signal')
axes.set_xlabel("T(s)");
axes.set_ylabel("Amplitude");
axes.set_title("Kernel Ridge Regression Method")
axes.legend(loc=0);
```

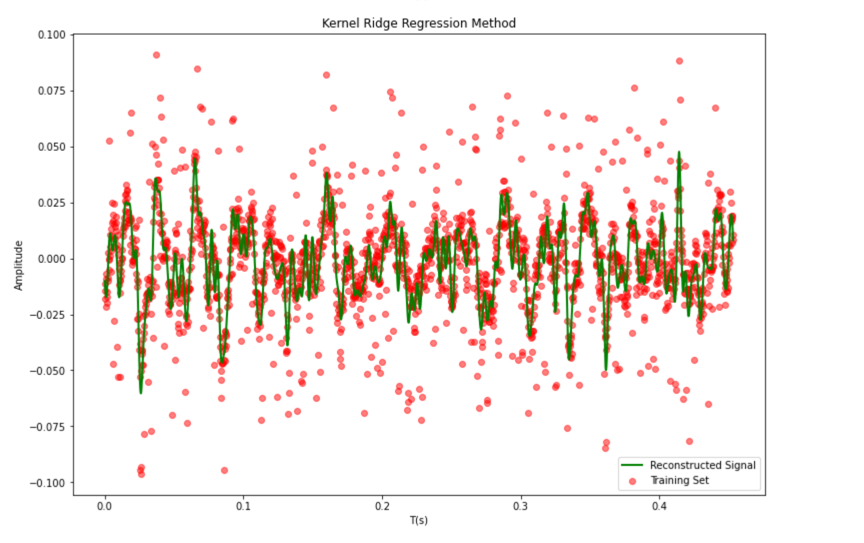


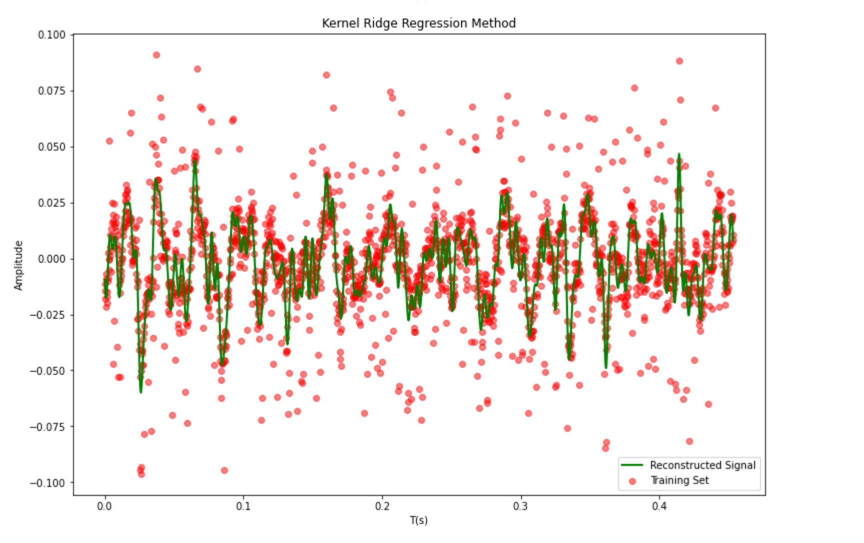
Part ii: Repeating step 'i' using new vlues for LAMDA

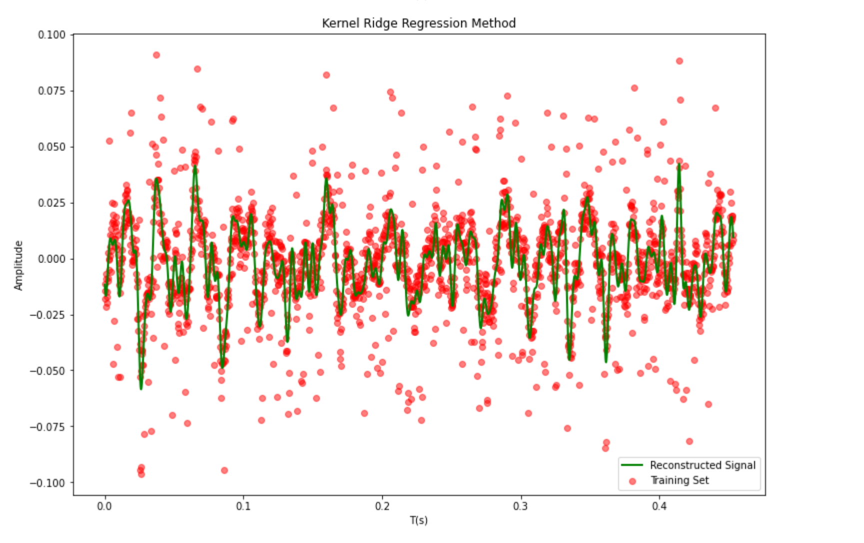
In [3]:
```python
SIGMA = 0.004
LAMDA = np.array([10**-6, 10**-5,0.0005, 0.001,0.01, 0.05])
for i in range (len(LAMDA)):
    Y_testing = Kernel_Ridge(x,y, x, SIGMA, LAMDA[i]) #Kernel Ridge predicted Y values.
    fig = plt.figure(figsize = (12, 8))
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    axes.scatter(x, y, color ='red'  , alpha = 0.5, label = "Training Set")
    axes.plot(x, Y_testing, "g-", linewidth = 2, label = 'Reconstructed Signal')
    axes.set_xlabel("T(s)");
    axes.set_ylabel("Amplitude");
    axes.set_title("Kernel Ridge Regression Method")
    axes.legend(loc=0);
```

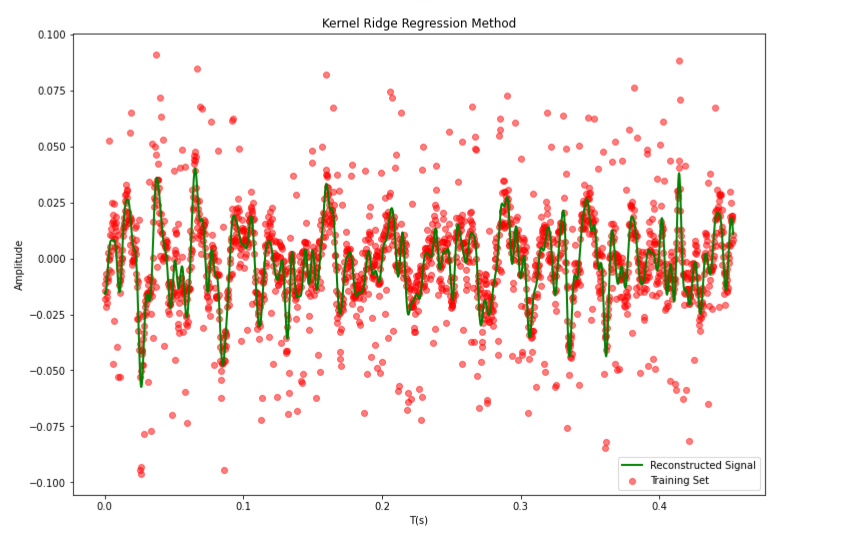






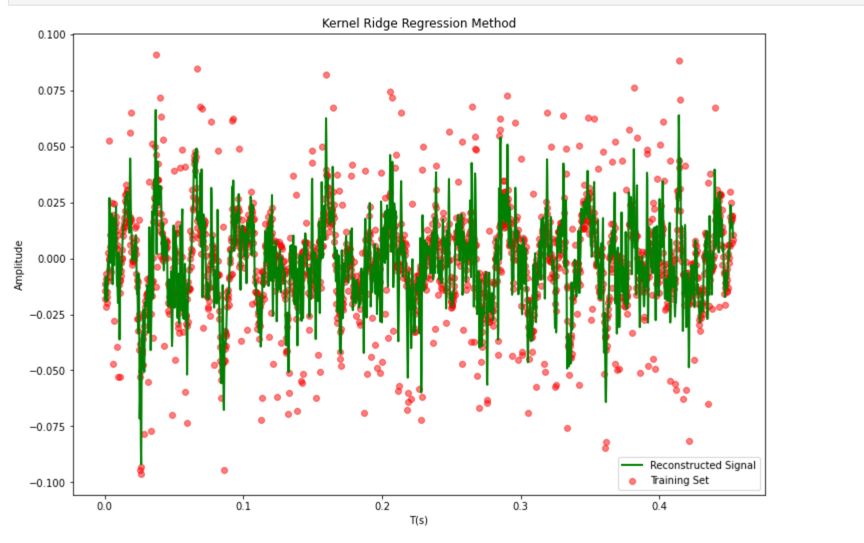


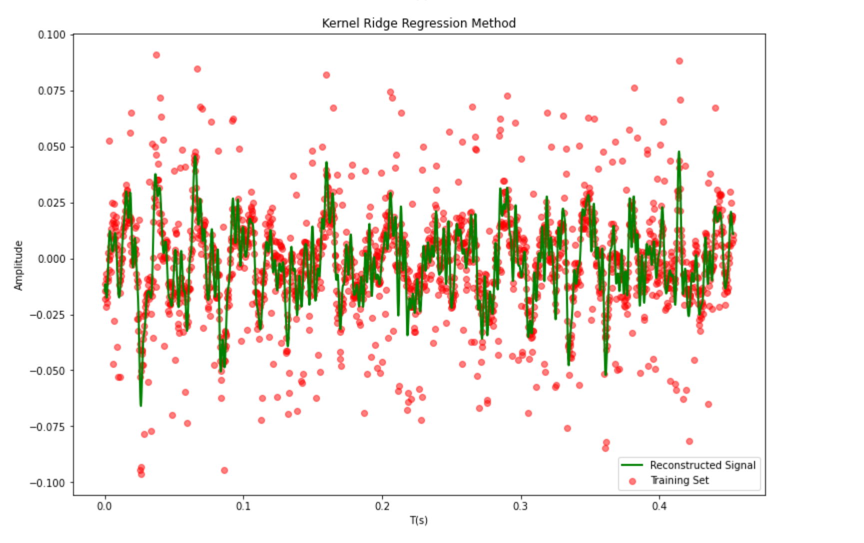


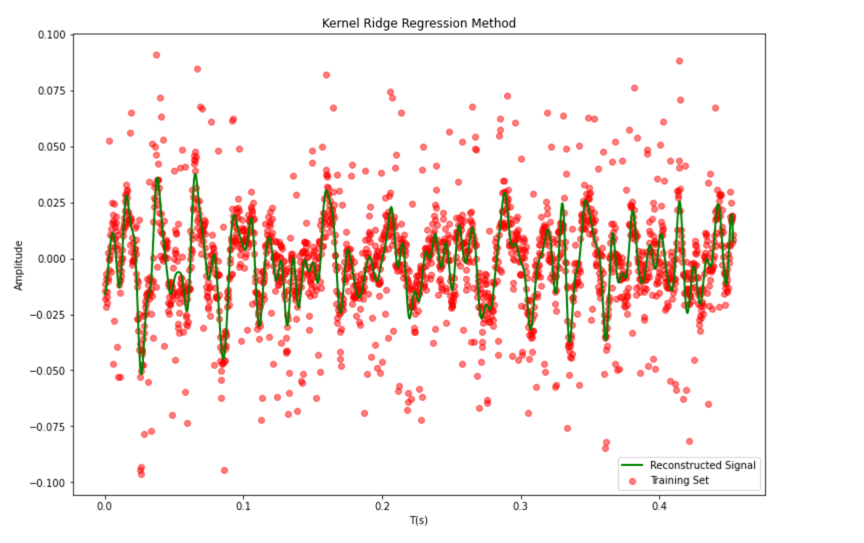


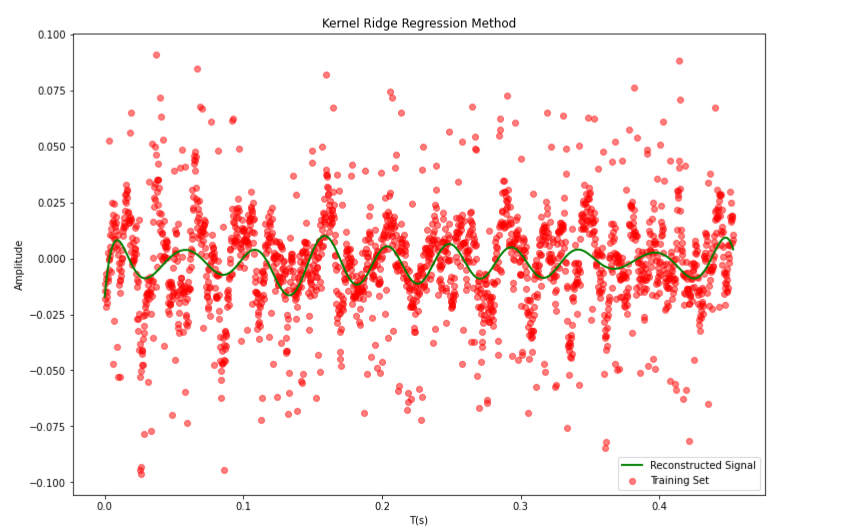
Part iii: Repeating step 'i' using new vlues for SIGMA

In [4]:
```python
LAMDA = 0.0001
SIGMA = np.array([0.001, 0.003,0.008, 0.05])
for i in range (len(SIGMA)):
    Y_testing = Kernel_Ridge(x,y, x, SIGMA[i], LAMDA) # Kernel Ridge predicted Y values.
    fig = plt.figure(figsize = (12, 8))
    axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])
    axes.scatter(x, y, color ='red'  , alpha = 0.5, label = "Training Set")
    axes.plot(x, Y_testing, "g-", linewidth = 2, label = 'Reconstructed Signal')
    axes.set_xlabel("T(s)");
    axes.set_ylabel("Amplitude");
    axes.set_title("Kernel Ridge Regression Method")
    axes.legend(loc=0);
```









Comment

*The value of Sigma is is effecting the graph sharply.

*According to the graph we can see that when the value of Sigma is high the model is underfits..

On the otherhand, when the value of Sigma is low the model overfits

*By increasing the value of lambda and sigmas, the quality of the reconstructed signal is degraded because outliers are missed.

In [ ]:

Assignment E: 5.3

In [ ]:

```python
import numpy as np
import scipy
import matplotlib.pyplot as plt
```

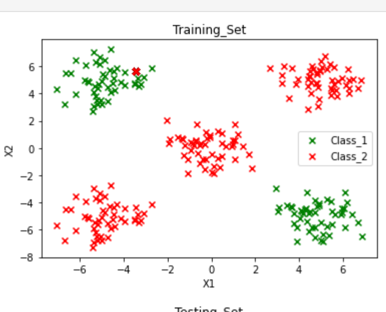
In [2]:

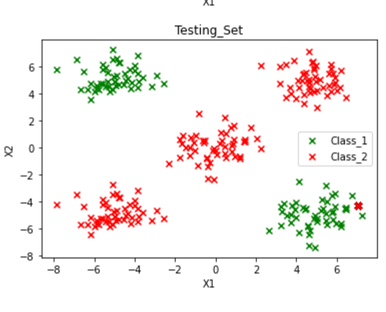
```python
def nn_model(X, Y, n_h, num_iterations , learning_rate , print_cost ):

    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 : (1, 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)]
```

Part i: Plotting data set

In [3]:

```python
seed = 7
D = 2  # Number of dimension
m1 = np.array([[-5, 5], [5, -5]]).T  # For class one, mean
m2 = np.array([[-5, -5], [0, 0], [5, 5]]).T # For class two, mean
c_1 = m1.shape[1]  # gaussians per class 1
c_2 = m2.shape[1]  # gaussians per class 2

#Training Set Generation
N_1 = np.array([50,50])
S1 = np.zeros(shape=(D,D, c_1))
s_1=1 #Variance
for i in range(0, c_1):
    S1[:, :, i] = np.array(s_1*np.eye(2))

w1_training = mixt_model(m1, S1, N_1, seed) #training set for first class
N_2 = np.array([50,50,50])
S2 = np.zeros(shape=(D, D, c_2))
for i in range(0, c_2):
    S2[:, :, i] = s_1 * np.eye(D)
w2_training= mixt_model(m2, S2, N_2, seed) #training set for second class
X_training = np.concatenate((w1_training,  w2_training), axis=1);
Y_training = np.concatenate((np.zeros(shape=(1, np.sum(N_1))), np.ones(shape=(1, np.sum(N_2)))), axis=1)

#Test Set Generation
seed = 5
w1_testing = mixt_model(m1, S1, N_1, seed)#test set for first class
w2_testing= mixt_model(m2, S2, N_2, seed) #test set for second class
X_testing= np.concatenate((w1_testing,  w2_testing), axis=1);
Y_testing=np.concatenate((np.zeros(shape=(1, np.sum(N_1))), np.ones(shape=(1, np.sum(N_2)))), axis=1)

plt.figure(1)
plt.scatter(X_training[0,np.nonzero(Y_training == 0)], X_training[1,np.nonzero(Y_training ==0)]
            , marker = "x", color = "g", label = "Class_1");
plt.scatter(X_training[0,np.nonzero(Y_training == 1)], X_training[1,np.nonzero(Y_training ==1)]
            , marker = "x", color = "r", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Training_Set");
plt.figure(2)
plt.scatter(X_testing[0,np.nonzero(Y_training == 0)], X_testing[1,np.nonzero(Y_training ==0)]
            , marker = "x", color = "g", label = "Class_1");
plt.scatter(X_testing[0,np.nonzero(Y_training == 1)], X_testing[1,np.nonzero(Y_training ==1)]
            , marker = "x", color = "r", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Testing_Set");
```



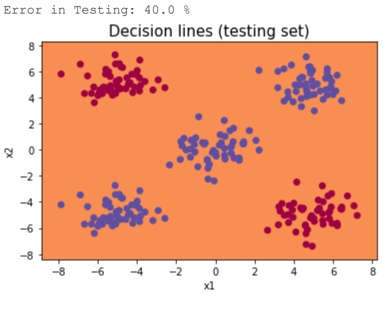


Part ii: Cmputing training, test errors and plotting decision boundaries

In [4]:

```python
paramets_training = nn_model(X_training,Y_training,n_h = 2,num_iterations = 9000,learning_rate = 0.01,print_cos

#Computing errors in Training and Testing
Y_predict_training = nn_predict(paramets_training, X_training)
print("Error in Training: {} %".format(np.mean(np.abs(Y_predict_training-Y_training))*100))

parameters_testing = nn_model(X_testing,Y_testing,n_h = 2,num_iterations = 9000,learning_rate = 0.01,print_cost
Y_predict_testing = nn_predict(parameters_testing, X_testing)
print("Error in Testing: {} %".format(np.mean(np.abs(Y_predict_testing-Y_testing))*100))

#Creating decision Boundaries established as a result of the network
plot_decision_boundary(lambda x:nn_predict(parameters_testing,x.T),X_testing,Y_testing.ravel());
plt.title("Decision lines (testing set)", fontsize=15);
```

```
Cost after iteration 0: 0.600423
Cost after iteration 1000: 0.239990
Cost after iteration 2000: 0.239988
Cost after iteration 3000: 0.239986
Cost after iteration 4000: 0.239985
Cost after iteration 5000: 0.239983
Cost after iteration 6000: 0.239980
Cost after iteration 7000: 0.239977
Cost after iteration 8000: 0.239972
Error in Training: 40.0 %
Cost after iteration 0: 0.600235
Cost after iteration 1000: 0.239998
Cost after iteration 2000: 0.239997
Cost after iteration 3000: 0.239996
Cost after iteration 4000: 0.239995
Cost after iteration 5000: 0.239994
Cost after iteration 6000: 0.239993
Cost after iteration 7000: 0.239990
Cost after iteration 8000: 0.239985
Error in Testing: 40.0 %
```

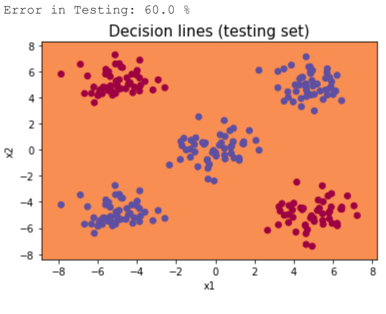


Part iii: Repeating step ii with step-size 0.0001

In [5]:

```python
paramets_training = nn_model(X_training,Y_training,n_h = 2,num_iterations = 9000,learning_rate = 0.0001,print_c

#Computing errors in Training and Testing
Y_predict_training = nn_predict(paramets_training, X_training)
print("Error in Training: {} %".format(np.mean(np.abs(Y_predict_training-Y_training))*100))

parameters_testing = nn_model(X_testing,Y_testing,n_h = 2,num_iterations = 9000,learning_rate = 0.0001,print_co
Y_predict_testing = nn_predict(parameters_testing, X_testing)
print("Error in Testing: {} %".format(np.mean(np.abs(Y_predict_testing-Y_testing))*100))

#Creating decision Boundaries established as a result of the network
plot_decision_boundary(lambda x:nn_predict(parameters_testing,x.T),X_testing,Y_testing.ravel());
plt.title("Decision lines (testing set)", fontsize=15);
```

```
Cost after iteration 0: 0.600423
Cost after iteration 1000: 0.535113
Cost after iteration 2000: 0.481649
Cost after iteration 3000: 0.437882
Cost after iteration 4000: 0.402051
Cost after iteration 5000: 0.372717
Cost after iteration 6000: 0.348701
Cost after iteration 7000: 0.329038
Cost after iteration 8000: 0.312939
Error in Training: 60.0 %
Cost after iteration 0: 0.600235
Cost after iteration 1000: 0.534994
Cost after iteration 2000: 0.481584
Cost after iteration 3000: 0.437857
Cost after iteration 4000: 0.402059
Cost after iteration 5000: 0.372749
Cost after iteration 6000: 0.348751
Cost after iteration 7000: 0.329102
Cost after iteration 8000: 0.313012
Error in Testing: 60.0 %
```



Comment

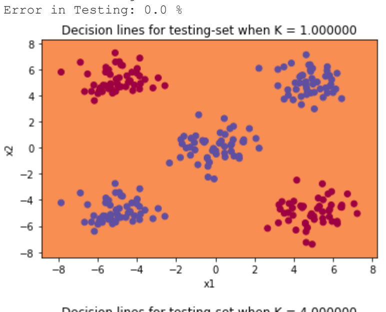
If the nodes are 1 or 2 then it is not classifying the points

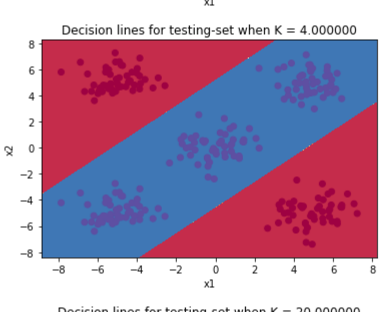
Part iv: Repeating step ii with k = 1,4,20 hidden layer nodes

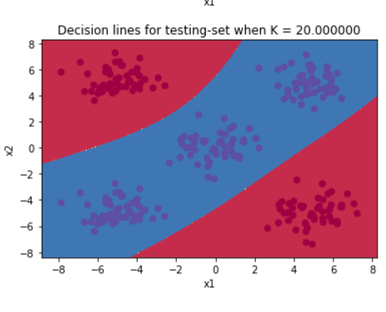
In [6]:

```python
n_h = np.array([1, 4, 20])
for i in range (len(n_h)):
    paramets_training = nn_model(X_training,Y_training,n_h[i],num_iterations = 9000,learning_rate = 0.01,print_

    #Computing errors in Training and Testing
    Y_predict_training = nn_predict(paramets_training, X_training)
    print("Error in Training: {} %".format(np.mean(np.abs(Y_predict_training-Y_training))*100))

    parameters_testing = nn_model(X_testing,Y_testing,n_h[i],num_iterations = 9000,learning_rate = 0.01,print_c
    Y_predict_testing = nn_predict(parameters_testing, X_testing)
    print("Error in Testing: {} %".format(np.mean(np.abs(Y_predict_testing-Y_testing))*100))
    plt.figure(i)
    #Creating decision Boundaries established as a result of the network
    plot_decision_boundary(lambda x:nn_predict(parameters_testing,x.T),X_testing,Y_testing.ravel());
    plt.title("Decision lines for testing-set when K = %f" %n_h[i], fontsize=12);
```

```
Error in Training: 40.0 %
Error in Testing: 40.0 %
Error in Training: 0.0 %
Error in Testing: 0.0 %
Error in Training: 0.0 %
Error in Testing: 0.0 %
```





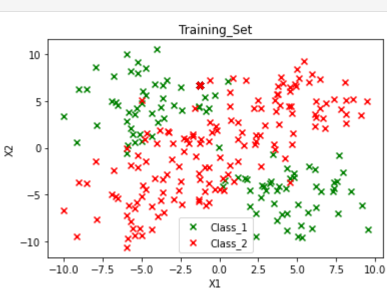


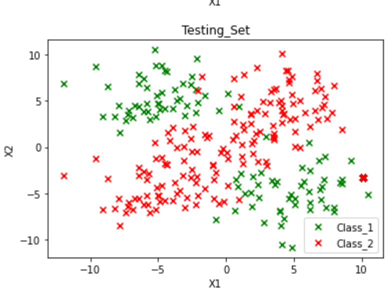
**** If the learning rate is too low then it can not classify the points because it can not reach its result¶**

Part v: Repeating step i-iii with different covariance matrix

In [7]:

```python
seed = 7
D = 2  # Number of dimension
m1 = np.array([[-5, 5], [5, -5]]).T  # For class one, mean
m2 = np.array([[-5, -5], [0, 0], [5, 5]]).T # For class two, mean
c_1 = m1.shape[1]  # gaussians per class 1
c_2 = m2.shape[1]  # gaussians per class 2

#Training Set Generation
N_1 = np.array([50,50])
S1 = np.zeros(shape=(D,D, c_1))
s_1= 6 #Variance
for i in range(0, c_1):
    S1[:, :, i] = np.array(s_1*np.eye(2))

w1_training = mixt_model(m1, S1, N_1, seed) #training set for first class
N_2 = np.array([50,50,50])
S2 = np.zeros(shape=(D, D, c_2))
for i in range(0, c_2):
    S2[:, :, i] = s_1 * np.eye(D)
w2_training= mixt_model(m2, S2, N_2, seed) #training set for second class
X_training = np.concatenate((w1_training,  w2_training), axis=1);
Y_training = np.concatenate((np.zeros(shape=(1, np.sum(N_1))), np.ones(shape=(1, np.sum(N_2)))), axis=1)

#Test Set Generation
seed = 5
w1_testing = mixt_model(m1, S1, N_1, seed)#test set for first class
w2_testing= mixt_model(m2, S2, N_2, seed) #test set for second class
X_testing= np.concatenate((w1_testing,  w2_testing), axis=1);
Y_testing=np.concatenate((np.zeros(shape=(1, np.sum(N_1))), np.ones(shape=(1, np.sum(N_2)))), axis=1)

plt.figure(1)
plt.scatter(X_training[0,np.nonzero(Y_training == 0)], X_training[1,np.nonzero(Y_training ==0)]
            , marker = "x", color = "g", label = "Class_1");
plt.scatter(X_training[0,np.nonzero(Y_training == 1)], X_training[1,np.nonzero(Y_training ==1)]
            , marker = "x", color = "r", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Training_Set");
plt.figure(2)
plt.scatter(X_testing[0,np.nonzero(Y_training == 0)], X_testing[1,np.nonzero(Y_training ==0)]
            , marker = "x", color = "g", label = "Class_1");
plt.scatter(X_testing[0,np.nonzero(Y_training == 1)], X_testing[1,np.nonzero(Y_training ==1)]
            , marker = "x", color = "r", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Testing_Set");
```





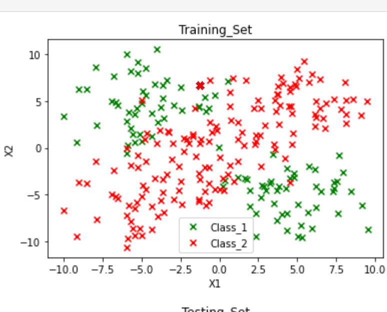
In [8]:

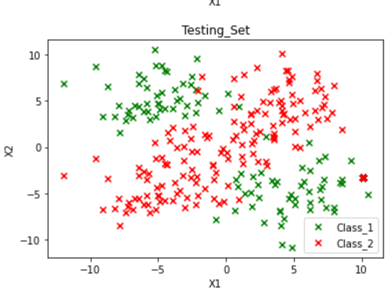
```python
n_h = np.array([2, 20, 50])
for i in range (len(n_h)):
    paramets_training = nn_model(X_training,Y_training,n_h[i],num_iterations = 9000,learning_rate = 0.01,print_

    #Computing errors in Training and Testing
    Y_predict_training = nn_predict(paramets_training, X_training)
    print("Error in Training: {} %".format(np.mean(np.abs(Y_predict_training-Y_training))*100))

    parameters_testing = nn_model(X_testing,Y_testing,n_h[i],num_iterations = 9000,learning_rate = 0.01,print_c
    Y_predict_testing = nn_predict(parameters_testing, X_testing)
    print("Error in Testing: {} %".format(np.mean(np.abs(Y_predict_testing-Y_testing))*100))
    plt.figure(i)
    #Creating decision Boundaries established as a result of the network
    plot_decision_boundary(lambda x:nn_predict(parameters_testing,x.T),X_testing,Y_testing.ravel());
    plt.title("Decision lines for testing-set when K = %f" %n_h[i], fontsize=12);
```

```
Error in Training: 40.0 %
Error in Testing: 40.0 %
Error in Training: 6.800000000000001 %
Error in Testing: 5.6000000000000005 %
Error in Training: 7.6 %
Error in Testing: 5.6000000000000005 %
```

In [9]:

```python
seed = 7
D = 2  # Number of dimension
m1 = np.array([[-5, 5], [5, -5]]).T  # For class one, mean
m2 = np.array([[-5, -5], [0, 0], [5, 5]]).T # For class two, mean
c_1 = m1.shape[1]  # gaussians per class 1
c_2 = m2.shape[1]  # gaussians per class 2

#Training Set Generation
N_1 = np.array([50,50])
S1 = np.zeros(shape=(D,D, c_1))
s_1= 6 #Variance
for i in range(0, c_1):
    S1[:, :, i] = np.array(s_1*np.eye(2))

w1_training = mixt_model(m1, S1, N_1, seed) #training set for first class
N_2 = np.array([50,50,50])
S2 = np.zeros(shape=(D, D, c_2))
for i in range(0, c_2):
    S2[:, :, i] = s_1 * np.eye(D)
w2_training= mixt_model(m2, S2, N_2, seed) #training set for second class
X_training = np.concatenate((w1_training,  w2_training), axis=1);
Y_training = np.concatenate((np.zeros(shape=(1, np.sum(N_1))), np.ones(shape=(1, np.sum(N_2)))), axis=1)

#Test Set Generation
seed = 5
w1_testing = mixt_model(m1, S1, N_1, seed)#test set for first class
w2_testing= mixt_model(m2, S2, N_2, seed) #test set for second class
X_testing= np.concatenate((w1_testing,  w2_testing), axis=1);
Y_testing=np.concatenate((np.zeros(shape=(1, np.sum(N_1))), np.ones(shape=(1, np.sum(N_2)))), axis=1)

plt.figure(1)
plt.scatter(X_training[0,np.nonzero(Y_training == 0)], X_training[1,np.nonzero(Y_training ==0)]
            , marker = "x", color = "g", label = "Class_1");
plt.scatter(X_training[0,np.nonzero(Y_training == 1)], X_training[1,np.nonzero(Y_training ==1)]
            , marker = "x", color = "r", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Training_Set");
plt.figure(2)
plt.scatter(X_testing[0,np.nonzero(Y_training == 0)], X_testing[1,np.nonzero(Y_training ==0)]
            , marker = "x", color = "g", label = "Class_1");
plt.scatter(X_testing[0,np.nonzero(Y_training == 1)], X_testing[1,np.nonzero(Y_training ==1)]
            , marker = "x", color = "r", label = "Class_2");
plt.legend(loc=0);
plt.xlabel("X1");
plt.ylabel("X2");
plt.title("Testing_Set");
```





In [10]:

```python
n_h = np.array([2, 20, 50])
for i in range (len(n_h)):
    paramets_training = nn_model(X_training,Y_training,n_h[i],num_iterations = 9000,learning_rate = 0.0001,prir

    #Computing errors in Training and Testing
    Y_predict_training = nn_predict(paramets_training, X_training)
    print("Error in Training: {} %".format(np.mean(np.abs(Y_predict_training-Y_training))*100))

    parameters_testing = nn_model(X_testing,Y_testing,n_h[i],num_iterations = 9000,learning_rate = 0.0001,print
    Y_predict_testing = nn_predict(parameters_testing, X_testing)
    print("Error in Testing: {} %".format(np.mean(np.abs(Y_predict_testing-Y_testing))*100))
    plt.figure(i)
    #Creating decision Boundaries established as a result of the network
    plot_decision_boundary(lambda x:nn_predict(parameters_testing,x.T),X_testing,Y_testing.ravel());
    plt.title("Decision lines for testing-set when K = %f" %n_h[i], fontsize=12);
```

```
Error in Training: 60.0 %
Error in Testing: 60.0 %
Error in Training: 60.0 %
Error in Testing: 60.0 %
Error in Training: 60.0 %
Error in Testing: 60.0 %
```