Machine Learning

Exercise 2: Linear/Quadratic Discriminant Analysis

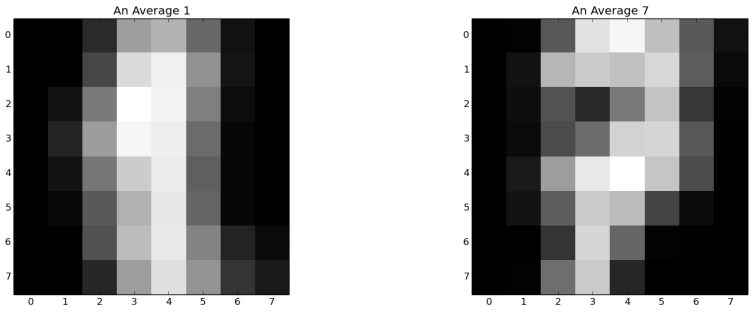
November 2, 2014

Group: Sergej Kraft
Elias Roeger
Ekaterina Tikhoncheva

Contents

1 Data Preparation

In this exercise we use the digit data set from *http://scikit-learn.org/stable/datasets/*. The main task is to write a classifier that should distinguish digit '1' from digit '7'.
The initial set of '1' and '7' consist of 361 Images, where each image is represented as a vector in 64-dimensional space. For purposes of this exercise we restrict ourself to 2-dimensional feature space. To do this we compute an average digit inside of each class (see image 1) and then the difference between this two average digits. We select two dimensions where the calculated difference has two biggest values.
To decide if the selected features are suitable for classification we draw a distribution of data points in the feature space (see image 2).
We can see that the points of the different classes excluding several points are good separated.



(a) An average '1'

(b) An average '7'

Figure 1: Average digits inside each class

For realisation of dimension reduction see function $dr(x, y)$, where x is the initial set of data points and y is an array of corresponding labels. The function $scatterplot(x, y, labels, title, imageName)$ implements the drawing of a scatter plot of the points.

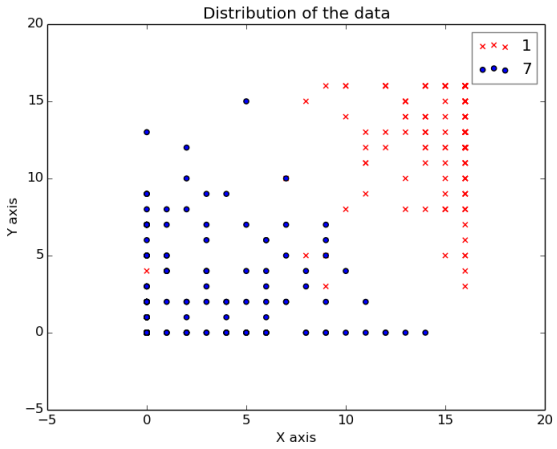


Figure 2: Distribution of the data points in the feature space

2 Nearest Mean Classifier

Implementation of the Nearest Mean Classifier

NearestMean.py

```
def NearestMean(x_training, y_training, x_test):

    nTr, dTr = x_training.shape
    nTest, dTest = x_test.shape

    assert dTr==dTest, 'Images in training and test sets have different size'

    labels = np.unique(y_training)
    nClasses= len(labels)

    # Compute class mean
    class_average = np.zeros((nClasses, dTr) , dtype = np.float16)
    for l in range(0,nClasses):
        # for each class
        ind = (y_training==labels[l])
        class_l = x_training[ind,:]
        nL = len(class_l) # number of elements in the class with label l
        class_average[l,:] = np.sum(class_l[:,:], axis = 0)/float(nL)

    # end for l

    prediction = np.zeros(nTest, dtype = np.int8)

    for i in range(0,nTest):
```

```
        # for each test image
        dist = class_average - x_test[i,:]
        dist = np.sqrt(np.sum(np.square(dist), axis = 1))

        min_dist = np.argmin(dist)
        prediction[i] = labels[min_dist]
    # end for i
    return prediction
# end NearestMean
```

Our implementation of the Nearest Mean Classifier provides the classification rate 99.31%.

3 Quadratic Discriminant Analysis (QDA)

Our implementation of the QDA - Training to compute the mean vectors, covariance matrices and priors of the classes.

QDAtraining.py

```
def compute_qda(trainingy, trainingx):
    # size(trainingx) = n1 x d
    # size(trainingy) = n1
    # d = 2
    d = trainingx.shape[1]

    # select element from each class
    x0 = trainingx[trainingy==0, :]        # Class of 0 (digit 1)
    n0 = x0.shape[0]

    x1 = trainingx[trainingy==1, :]    # Class of 1 (digit 7)
    n1 = x1.shape[0]

    # Compute the class means
    mu0 = np.sum(x0[:,:], axis = 0)/float(n0)
    mu1 = np.sum(x1[:,:], axis = 0)/float(n1)

    # Compute the covariance matrices

    covmat0 = np.zeros((d, d), dtype = np.float32);
    for i in range(0,d):
        for j in range(0,d):
            covmat0[i,j] = np.dot(x0[:,i]-mu0[i], x0[:,j]-mu0[j])/x0.shape[0]
        # end for j
    # end for i

    covmat1 = np.zeros((d, d), dtype = np.float32);
    for i in range(0,d):
        for j in range(0,d):
            covmat1[i,j] = np.dot(x1[:,i]-mu1[i], x1[:,j]-mu1[j])/x1.shape[0]
```

```
        # end for j
    # end for i

    #Compute the priors
    p0 = float(n0)/(n0+n1)
    p1 = float(n1)/(n0+n1)

    return mu0,mu1,covmat0,covmat1,p0,p1
# end compute_qda
```

Our implementation of the QDA - Prediction :

QDAprediction.py

```
# QDA Prediction
def perform_qda(mu0, mu1, covmat0, covmat1, p0, p1, testx):
    n2 = testx.shape[0]

    qda_predict = np.zeros(n2, dtype = np.int8)
    for i in range(0,n2):

        # k =0
        b_0 = -np.log(np.linalg.det(2*np.pi*covmat0))/2. - np.log(p0)

        covmat0_inv = np.linalg.inv(covmat0)

        testx_centered0 = testx - mu0

        y0 =  - np.dot ( np.dot(testx_centered0[i], covmat0_inv), \
                         testx_centered0[i].T)/2. - b_0

        # k =1
        b_1 = -np.log(np.linalg.det(2*np.pi*covmat1))/2. - np.log(p1)
        covmat1_inv = np.linalg.inv(covmat1)

        testx_centered1 = testx - mu1

        y1 = - np.dot ( np.dot(testx_centered1[i], covmat1_inv), \
                        testx_centered1[i].T)/2. - b_1

        # argmax (y0, y1)
        if y1>y0 :
            qda_predict[i] = 1
        else:
            qda_predict[i] = 0
        # end if
    # end for i

    return qda_predict
#end perform_qda
```

The run test on the training and test sets are present in the Table 1, the decision boundary of QDA is visualised on the image 3. We also visualised the results of the classification. The correct classified points are marked with bigger symbols (see image 4).
We can see that the QDA has a higher classification rate. The misclassifications stem from the not perfect distribution of the set points in the feature space : some point of one class lie between the points of the other class and thus on the wrong side of the decision boundary.

Set	Correct Classification rate
training set	97.22%
test set	99.31%

Table 1: Correct Classification Rate of QDA

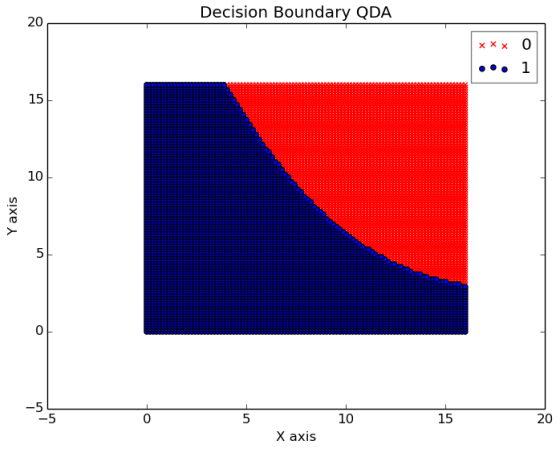
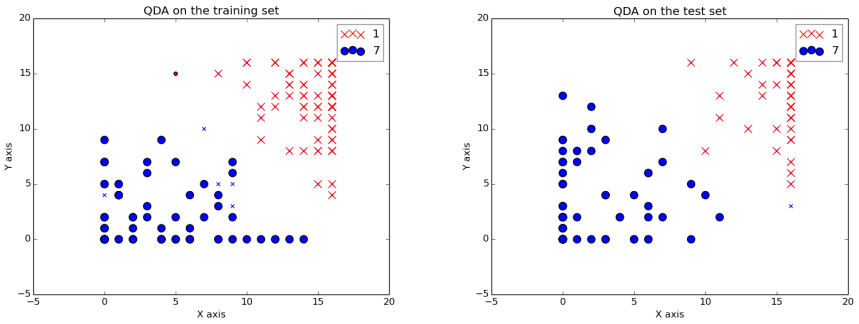


Figure 3: Decision boundary of the QDA



(a) Training set (b) Test set

Figure 4: Results of QDA Prediction

4 Linear Discriminant Analysis (LDA)

Implementation of the LDA prediction:

LDAprediction.py

```
# LDA Prediction
def perform_lda(mu0, mu1, covmat0, covmat1, p0, p1, testx):
    n2 = testx.shape[0]

    lda_predict = np.zeros(n2, dtype = np.int8)
    for i in range(0,n2):

        # k =0
        b_0 = - np.log(np.linalg.det(2*np.pi*covmat0))/2. - np.log(p0)

        covmat0_inv = np.linalg.inv(covmat0)

        w_0 = np.dot(covmat0_inv, mu0.T)

        b_0 = -b_0 - np.dot( mu0, w_0) /2.

        y0 = np.dot( testx[i], w_0) + b_0

        # k =1
        b_1 = - np.log(np.linalg.det(2*np.pi*covmat1))/2. - np.log(p1)

        covmat1_inv = np.linalg.inv(covmat1)

        w_1 = np.dot(covmat1_inv, mu1.T)

        b_1 += np.dot( mu1, w_1) /2.
```

```
        y1 = - np.dot( testx[i], w_1) - b_1

        # argmax (y0, y1)
        if y1>y0 :
            lda_predict[i] = 1
        else:
            lda_predict[i] = 0
        # end if
    # end for i

    return lda_predict
#end perform_lda
```

The run test on the training and test sets are present in the Table 2, the decision boundary of LDA is visualised on the image 5. The results of the classification are presented on the image 6.
One can see, that LDA provides lower classification rate as QDA.

Set	Correct Classification rate
training set	93.06%
test set	95.86%

Table 2: Correct Classification Rate of LDA

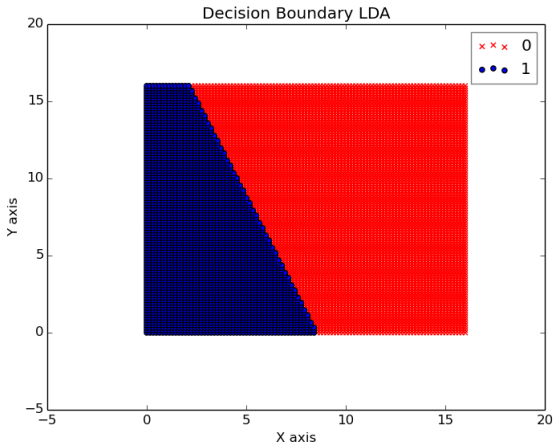
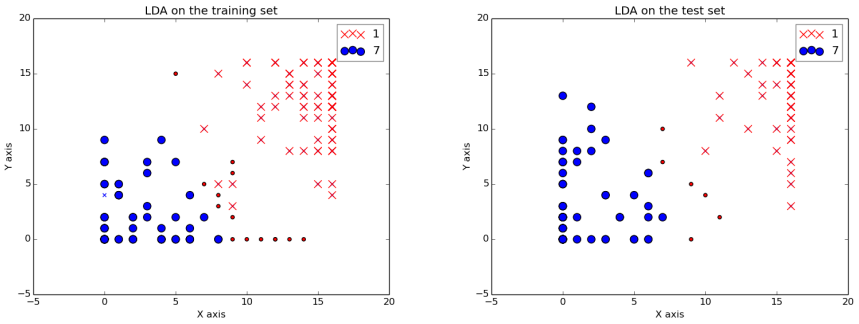


Figure 5: Decision boundary of the LDA



(a) Training set (b) Test set

Figure 6: Results of LDA Prediction

5 Complete Code

../ex02.py

```
"""
Exercise 2 : LDA/QDA and the Nearest Mean Classifier

"""

import numpy as np
import matplotlib.pyplot as plot

from sklearn.datasets import load_digits
from sklearn import cross_validation
# ------------------------------------------------------------------------
# Dimension Reduction Function
# size(x) = n x d.
# size(dr(x)) = n x 2
def dr(x, y):

    # Calculate the average digit 1
    x_1 = x[y==1]
    n1 = len(x_1)
    average1 = np.sum(x_1[:,:], axis = 0)/float(n1)

#     # Show average 1
#     f = plot.figure()
#     plot.title('An Average 1')
#     plot.gray()
#     plot.imshow(np.reshape(average1, (8,8)), interpolation = 'nearest');
#     plot.show()
#     f.savefig('average1.png')
```

```
    # Calculate the average digit 7
    x_7 = x[y==7]
    n7 = len(x_7)
    average7 = np.sum(x_7[:,:], axis = 0)/float(n7)
#    # Show average 7
#    f = plot.figure()
#    plot.title('An Average 7')
#    plot.gray()
#    plot.imshow(np.reshape(average7, (8,8)), interpolation = 'nearest');
#    plot.show()
#    f.savefig('average7.png')


    # Differences between average1 and average7
    diff17 = np.abs(average1-average7)

    # Sort in descending order
    diff_sortInd = np.argsort(diff17);
    diff_sortInd = diff_sortInd[::-1]

    # leave only indices of the two first elements
    # It means, we choose two dimensions, where average digits have highest
    # difference
    diff_sortInd = diff_sortInd[0:2:1]

    xn = x[:,diff_sortInd]

    return xn

#end def dr

# ------------------------------------------------------------------------
# 2 Nearest Mean Classifier
def NearestMean(x_training, y_training, x_test):

    nTr, dTr = x_training.shape
    nTest, dTest = x_test.shape

    assert dTr==dTest, 'Images in training and test sets have different size'

    labels = np.unique(y_training)
    nClasses= len(labels)

    # Compute class mean
    class_average = np.zeros((nClasses, dTr) , dtype = np.float16)
    for l in range(0,nClasses):
        # for each class
        ind = (y_training==labels[l])
        class_l = x_training[ind,:]
        nL = len(class_l) # number of elements in the class with label l
```

```
            class_average[l,:] = np.sum(class_l[:,:], axis = 0)/float(nL)

        # end for l

        prediction = np.zeros(nTest, dtype = np.int8)

        for i in range(0,nTest):
            # for each test image
            dist = class_average - x_test[i,:]
            dist = np.sqrt(np.sum(np.square(dist), axis = 1))

            min_dist = np.argmin(dist)
            prediction[i] = labels[min_dist]
        # end for i
        return prediction
    # end NearestMean

    # ------------------------------------------------------------------------
    ## Draw a scatter plot of sets of points x1, x2
    # x_i in R^(n_i,2), i=1,2

    def scatterplot(x,y, labels = [0,1], title = 'Feature Space',\
                                                imageName = 'scatterplot.png'):

        x_0 = x[y==labels[0]]
        x_1 = x[y==labels[1]]

        f = plot.figure()
        plot.title(title)
        plot.xlabel('X axis')
        plot.ylabel('Y axis')
        plot.scatter(x_0[:,0],x_0[:,1], marker="x", c="r", label = labels[0])
        plot.scatter(x_1[:,0],x_1[:,1], marker="o", c="b", label = labels[1])
        plot.legend(framealpha=0.5)

        plot.show()
        f.savefig(imageName)
    #end def

    # ------------------------------------------------------------------------
    ## Draw a scatter plot of sets of points x1, x2
    # x_i in R^(n_i,2), i=1,2

    def plotPredictionResults(x,y_test, y_predict, title = 'Prediction Results',\
                              imageName = 'predictionResults.png'):

        x_0 = x[y_test==0]

        x_1 = x[y_test==1]
```

```
    x_0_correct = x[np.logical_and(y_test == 0, y_predict == 0)]
    x_1_correct = x[np.logical_and(y_test == 1, y_predict == 1)]



    f = plot.figure()
    plot.title(title)
    plot.xlabel('X axis')
    plot.ylabel('Y axis')

    plot.scatter(x_0[:,0],x_0[:,1], marker="x", c="b")
    plot.scatter(x_1[:,0],x_1[:,1], marker="o", c="r")

    # mark correct predicted point with a bigger symbol
    plot.scatter(x_0_correct[:,0],x_0_correct[:,1], marker="x", s=90,\
                                                    c="r",label = '1')
    plot.scatter(x_1_correct[:,0],x_1_correct[:,1], marker="o", s=90,\
                                                    c="b",label = '7')

    plot.legend(framealpha=0.5)
    plot.show()

    f.savefig(imageName)
#end def
# ------------------------------------------------------------------------------

## QDA Training
# mu0, mu1 mean vectors
# covmat0, covmat1 covariance matrix
# p0, p1 priors (scalars)
def compute_qda(trainingy, trainingx):
    # size(trainingx) = n1 x d
    # size(trainingy) = n1
    # d = 2
    d = trainingx.shape[1]

    # select element from each class
    x0 = trainingx[trainingy==0, :]        # Class of 0 (digit 1)
    n0 = x0.shape[0]

    x1 = trainingx[trainingy==1, :]    # Class of 1 (digit 7)
    n1 = x1.shape[0]

    # Compute the class means
    mu0 = np.sum(x0[:,:], axis = 0)/float(n0)
    mu1 = np.sum(x1[:,:], axis = 0)/float(n1)

    # Compute the covariance matrices

    covmat0 = np.zeros((d, d), dtype = np.float32);
```

```
    for i in range(0,d):
        for j in range(0,d):
            covmat0[i,j] = np.dot(x0[:,i]-mu0[i], x0[:,j]-mu0[j])/x0.shape[0]
        # end for j
    # end for i

    covmat1 = np.zeros((d, d), dtype = np.float32);
    for i in range(0,d):
        for j in range(0,d):
            covmat1[i,j] = np.dot(x1[:,i]-mu1[i], x1[:,j]-mu1[j])/x1.shape[0]
        # end for j
    # end for i

    #Compute the priors
    p0 = float(n0)/(n0+n1)
    p1 = float(n1)/(n0+n1)

    return mu0,mu1,covmat0,covmat1,p0,p1
# end compute_qda

# ---------------------------------------------------------------------------
# QDA Prediction
def perform_qda(mu0, mu1, covmat0, covmat1, p0, p1, testx):
    n2 = testx.shape[0]

    qda_predict = np.zeros(n2, dtype = np.int8)
    for i in range(0,n2):

        # k =0
        b_0 = -np.log(np.linalg.det(2*np.pi*covmat0))/2. - np.log(p0)

        covmat0_inv = np.linalg.inv(covmat0)

        testx_centered0 = testx - mu0

        y0 =  - np.dot ( np.dot(testx_centered0[i], covmat0_inv), \
                         testx_centered0[i].T)/2. - b_0

        # k =1
        b_1 = -np.log(np.linalg.det(2*np.pi*covmat1))/2. - np.log(p1)
        covmat1_inv = np.linalg.inv(covmat1)

        testx_centered1 = testx - mu1

        y1 = - np.dot ( np.dot(testx_centered1[i], covmat1_inv), \
                        testx_centered1[i].T)/2. - b_1

        # argmax (y0, y1)
        if y1>y0 :
            qda_predict[i] = 1
```

```
        else:
            qda_predict[i] = 0
        # end if
    # end for i

    return qda_predict
#end perform_qda

# ----------------------------------------------------------------------------
## LDA Training
# mu0, mu1 mean vectors
# covmat0, covmat1 covariance matrix
# p0, p1 priors (scalars)
def compute_lda(trainingy, trainingx):
    # size(trainingx) = n1 x d
    # size(trainingy) = n1
    # d = 2
    d = trainingx.shape[1]

    # select element from each class
    x0 = trainingx[trainingy==0, :]         # Class of 0 (digit 1)
    n0 = x0.shape[0]

    x1 = trainingx[trainingy==1, :]     # Class of 1 (digit 7)
    n1 = x1.shape[0]

    # Compute the class means
    mu0 = np.sum(x0[:,:], axis = 0)/float(n0)
    mu1 = np.sum(x1[:,:], axis = 0)/float(n1)

    # Compute the covariance matrices

    covmat0 = np.zeros((d, d), dtype = np.float32);
    for i in range(0,d):
        for j in range(0,d):
            covmat0[i,j] = np.dot(x0[:,i]-mu0[i], x0[:,j]-mu0[j])/x0.shape[0]
        # end for j
    # end for i

    covmat1 = np.zeros((d, d), dtype = np.float32);
    for i in range(0,d):
        for j in range(0,d):
            covmat1[i,j] = np.dot(x1[:,i]-mu1[i], x1[:,j]-mu1[j])/x1.shape[0]
        # end for j
    # end for i

    #Compute the priors
    p0 = float(n0)/(n0+n1)
    p1 = float(n1)/(n0+n1)
```

```
    return mu0,mu1,covmat0,covmat1,p0,p1
# end compute_lda
# ------------------------------------------------------------------------------
# LDA Prediction
def perform_lda(mu0, mu1, covmat0, covmat1, p0, p1, testx):
    n2 = testx.shape[0]

    lda_predict = np.zeros(n2, dtype = np.int8)
    for i in range(0,n2):

        # k =0
        b_0 = - np.log(np.linalg.det(2*np.pi*covmat0))/2. - np.log(p0)

        covmat0_inv = np.linalg.inv(covmat0)

        w_0 = np.dot(covmat0_inv, mu0.T)

        b_0 = -b_0 - np.dot( mu0, w_0) /2.

        y0 = np.dot( testx[i], w_0) + b_0

        # k =1
        b_1 = - np.log(np.linalg.det(2*np.pi*covmat1))/2. - np.log(p1)

        covmat1_inv = np.linalg.inv(covmat1)

        w_1 = np.dot(covmat1_inv, mu1.T)

        b_1 += np.dot( mu1, w_1) /2.

        y1 = - np.dot( testx[i], w_1) - b_1

        # argmax (y0, y1)
        if y1>y0 :
            lda_predict[i] = 1
        else:
            lda_predict[i] = 0
        # end if
    # end for i

    return lda_predict
#end perform_lda
# ------------------------------------------------------------------------------
# Calculate the correct classification rate
# D  - labels set
# In our case D = [0,1]
def correctClassRate(y_pred, y_test, D, print_confMatrix = False):
    n = len(D)

    # calculate confusion matrix
```

```
    confusionM = np.zeros((n,n), dtype = np.float16)
    for i in range(0,n):
        # find positions of the digit n in test set

        indn = (y_test == D[i])
        # get predicted values on the corresponding positions
        predict = y_pred[indn]
        votes_bin = np.bincount(predict, minlength = 10)
        confusionM[i,:] = np.array(votes_bin[D])
    # end for-loop

    if print_confMatrix:
        print
        print 'Confusion Matrix '
        print confusionM
    # end if print_confMatrix

    # correct classification rate
    ccr = np.trace(confusionM)/len(y_test)
    return  ccr
# end correctClassRate

# ============================================================================
# ============================================================================

def main():
    plot.close('all')

#   Task 1 : Data Preparation

    print '-------------------------------------------------------------------'
    print ' 1 Data Preparation  '
    print '-------------------------------------------------------------------'

    digits = load_digits()
    print digits.keys()

    data = digits['data']
    target = digits['target']

    print 'Size of the whole digit set {}'. format(digits.data.shape)


    # we consider only 1s and 7s

    ind1 = (target==1)
    ind7 = (target==7)

    x_17 = data[ind1+ind7]
    y_17 = target[ind1+ind7]
```

```
n,d = x_17.shape
print 'Size of the set of 1s and 7s {}'. format(n)

# 1.1 Dimension Reduction

x_17 = dr(x_17, y_17)

# split the filtered data set in a training and test set
x_train, x_test, y_train, y_test = cross_validation.train_test_split(\
            x_17,y_17, train_size = 0.6, test_size = 0.4, random_state = 0)

nTrain = len(x_train)
nTest = len(x_test)


# 1.2 Scatterplot
# Draw distribution of the set in the feature space
scatterplot(x_17, y_17, [1,7], 'Distribution of the data', 'dataDistribution.png'

print
print '-------------------------------------------------------------------'
print ' 2 Nearest mean '
print '-------------------------------------------------------------------'

y_predict = NearestMean(x_train, y_train, x_test)
ccr_NM = correctClassRate(y_predict, y_test, [1,7],\
                                            print_confMatrix = False)
print 'Nearest Mean Correct Classification rate: {}'.format(ccr_NM)


print
print '-------------------------------------------------------------------'
print ' 3 QDA '
print '-------------------------------------------------------------------'

## 3.1 QDA - Training

# The new vectors of labels with values 0 or 1
# 0 corresponds to initial label 0
# 1 - to initial label 1
y_train01 = np.zeros(nTrain, dtype = np.int8)
y_train01[y_train==7] = 1 # 0<->1, 1<->7

y_test01 = np.zeros(nTest, dtype = np.int8)
y_test01[y_test==7] = 1 # 0<->1, 1<->7

# mu0, mu1 mean vectors
# covmat0, covmat1 covariance matrix
```

```
# p0, p1 priors (scalars)
mu0,mu1,covmat0,covmat1,p0,p1 = compute_qda(y_train01, x_train)

# 3.3 Apply the QDA prediction to the training set
print 'QDA prediction on the training set'
qda_predict_train = perform_qda(mu0, mu1, covmat0, covmat1, p0, p1, x_train)

# Compute the correct classification rate
ccr_train = correctClassRate(qda_predict_train, y_train01, [0,1],\
                                                print_confMatrix = True)
print 'Correct Classification rate on the training set:{}'.format(ccr_train)

# Visualize  the results of prediction
plotPredictionResults(x_train, y_train01, qda_predict_train,\
                        'QDA on the training set', 'QDAtrainingset.png');

# Visualize the decision boundary
# create grid of points
min_x_axis = np.min(x_train[:,0])
max_x_axis = np.max(x_train[:,0])

min_y_axis = np.min(x_train[:,1])
max_y_axis = np.max(x_train[:,1])

x_range = np.linspace(min_x_axis, max_x_axis, 100)
y_range = np.linspace(min_y_axis, max_y_axis, 100)

grid = np.dstack(np.meshgrid(x_range, y_range)).reshape(-1, 2)
# run QDA on grid
qda_predict_grid = perform_qda(mu0, mu1, covmat0, covmat1, p0, p1, grid )
# plot the results
scatterplot(grid, qda_predict_grid, [0,1], \
                        'Decision Boundary QDA','DecisionBoundaryQDA.png')


# Apply the QDA prediction to the test set
print
print 'QDA prediction on the test set'
qda_predict_test = perform_qda(mu0, mu1, covmat0, covmat1, p0, p1, x_test )

# Compute the correct classification rate
ccr_test = correctClassRate(qda_predict_test, y_test01, [0,1],\
                                                print_confMatrix = True)
print 'Correct Classification rate on the test set:{}'.format(ccr_test)

# Visualize  the results of prediction
plotPredictionResults(x_test, y_test01, qda_predict_test, \
                                'QDA on the test set', 'QDAtestset.png')

print '-----------------------------------------------------------------'
```

```
print ' 4 LDA '
print '-----------------------------------------------------------------'

## LDA - Training

# mu0, mu1 mean vectors
# covmat0, covmat1 covariance matrix
# p0, p1 priors (scalars)
mu0,mu1,covmat0,covmat1,p0,p1 = compute_lda(y_train01, x_train)

# 3.3 Apply the LDA prediction to the training set
print 'LDA prediction on the training set'
lda_predict_train = perform_lda(mu0, mu1, covmat0, covmat1, p0, p1, x_train)

# Compute the correct classification rate
ccr_train = correctClassRate(lda_predict_train, y_train01, [0,1],\
                                                print_confMatrix = True)
print 'Correct Classification rate on the training set:{}'.format(ccr_train)

# Visualize  the results of prediction
plotPredictionResults(x_train, y_train01, lda_predict_train,\
                'LDA on the training set', 'LDAtrainingset.png');

# Visualize the decision boundary
# create grid of points
min_x_axis = np.min(x_train[:,0])
max_x_axis = np.max(x_train[:,0])

min_y_axis = np.min(x_train[:,1])
max_y_axis = np.max(x_train[:,1])

x_range = np.linspace(min_x_axis, max_x_axis, 100)
y_range = np.linspace(min_y_axis, max_y_axis, 100)

grid = np.dstack(np.meshgrid(x_range, y_range)).reshape(-1, 2)
# run QDA on grid
lda_predict_grid = perform_lda(mu0, mu1, covmat0, covmat1, p0, p1, grid )

# plot the results
scatterplot(grid, lda_predict_grid, [0,1],\
                        'Decision Boundary LDA', 'DecisionBoundaryLDA.png')


# Apply the LDA prediction to the test set
print
print 'LDA prediction on the test set'
lda_predict_test = perform_lda(mu0, mu1, covmat0, covmat1, p0, p1, x_test )

# Compute the correct classification rate
ccr_test = correctClassRate(lda_predict_test, y_test01, [0,1],\
```

```
                                                    print_confMatrix = True)
    print 'Correct Classification rate on the test set:{}'.format(ccr_test)

    # Visualize  the results of prediction
    plotPredictionResults(x_test, y_test01, lda_predict_test, \
                          'LDA on the test set', 'LDAtestset.png')

    return 0

if __name__ == "__main__":
    main()
```