MACHINE LEARNING HOMEWORK 2

		Q1	Q2	Q3	Q4	Q5	Total
Grade	Max	1	1	1	1	1	5 points
	Expected	1	1	1	1	1	5

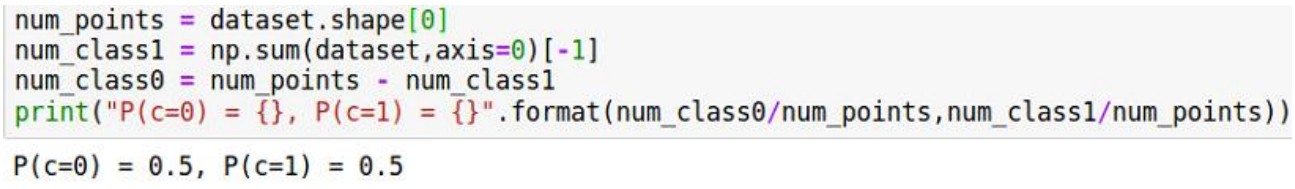
STUDENT

NAME SURNAME	: ALPEREN KANTARCI
STUDENT NUMBER	: 504191504
DEADLINE	: 10/11/2019
COURSE NAME	: MACHINE LEARNING
INSTRUCTOR	: YUSUF YASLAN

To test all the questions, “python HW2.py” command should be executed and dataset file “hw2.mat” data should be in the same folder with the script.

QUESTION 1:

Dataset contains 2000 data points with two features each with a binary class label. Labels are either 0 or 1. By executing following lines we can calculate the probability of a class.

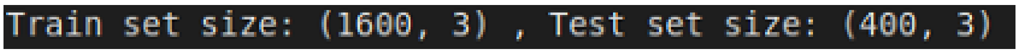


```
num_points = dataset.shape[0]
num_class1 = np.sum(dataset,axis=0)[-1]
num_class0 = num_points - num_class1
print("P(c=0) = {}, P(c=1) = {}".format(num_class0/num_points,num_class1/num_points))
```

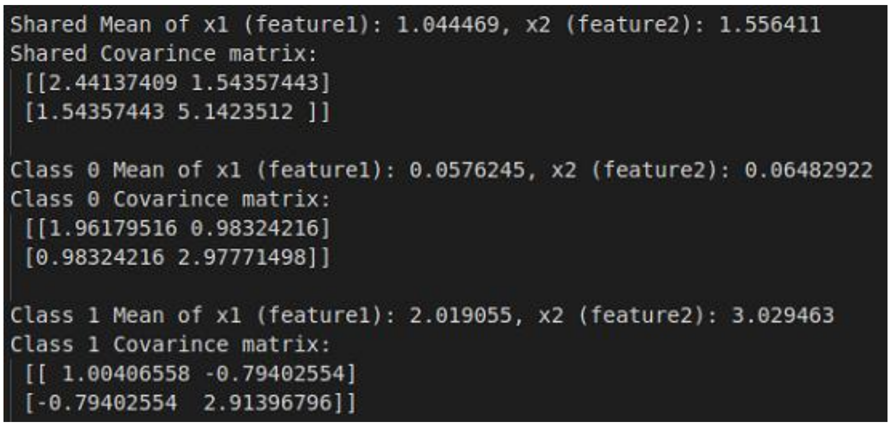
```
P(c=0) = 0.5, P(c=1) = 0.5
```

QUESTION 2:

To randomly select sets, i shuffled the dataset then dataset sliced into training and test set with 80% and 20% respectively. Then mean vectors and covariances of training set computed for each classes and all together. You can see the dataset sizes, means and covariances of the training set. These values also printed to terminal when script would run.

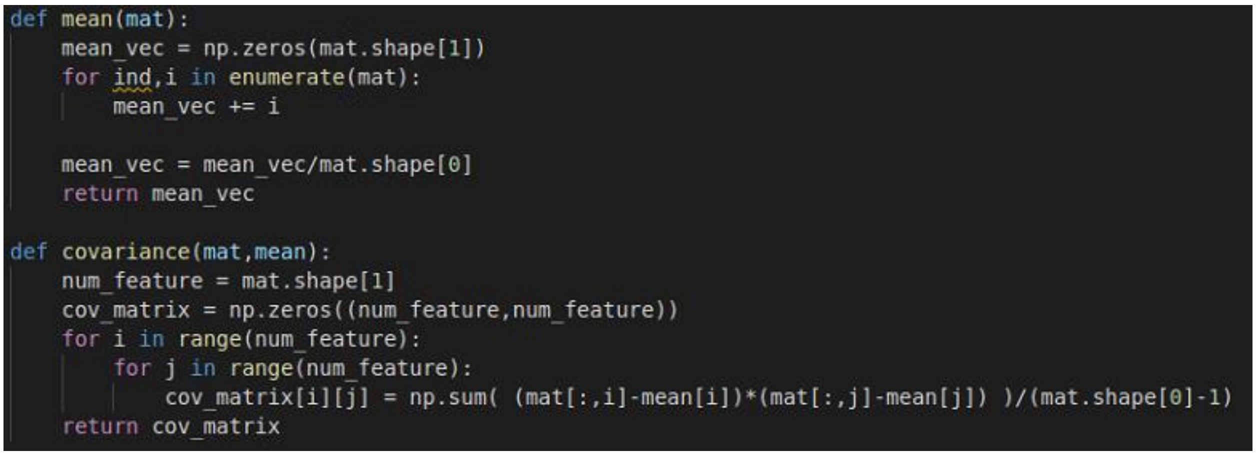


```
Train set size: (1600, 3) , Test set size: (400, 3)
```



```
Shared Mean of x1 (feature1): 1.044469, x2 (feature2): 1.556411
Shared Covarince matrix:
 [[2.44137409 1.54357443]
 [1.54357443 5.1423512 ]]

Class 0 Mean of x1 (feature1): 0.0576245, x2 (feature2): 0.06482922
Class 0 Covarince matrix:
 [[1.96179516 0.98324216]
 [0.98324216 2.97771498]]

Class 1 Mean of x1 (feature1): 2.019055, x2 (feature2): 3.029463
Class 1 Covarince matrix:
 [[ 1.00406558 -0.79402554]
 [-0.79402554  2.91396796]]
```

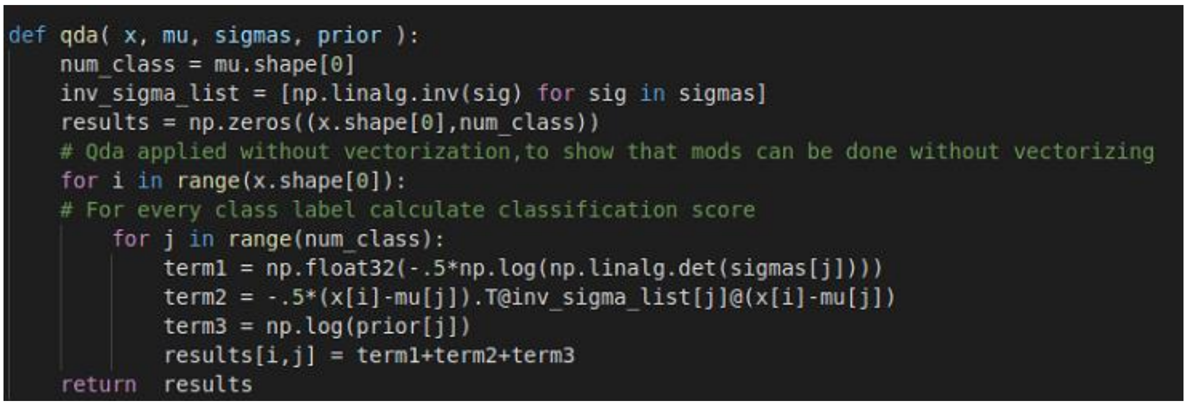
To compute the mean vector and covariance matrices, two functions implemented and each class given to the functions seperately. You can see the code screenshot of the calculation functions.



```python
def mean(mat):
    mean_vec = np.zeros(mat.shape[1])
    for ind,i in enumerate(mat):
        mean_vec += i

    mean_vec = mean_vec/mat.shape[0]
    return mean_vec

def covariance(mat,mean):
    num_feature = mat.shape[1]
    cov_matrix = np.zeros((num_feature,num_feature))
    for i in range(num_feature):
        for j in range(num_feature):
            cov_matrix[i][j] = np.sum( (mat[:,i]-mean[i])*(mat[:,j]-mean[j]) )/(mat.shape[0]-1)
    return cov_matrix
```

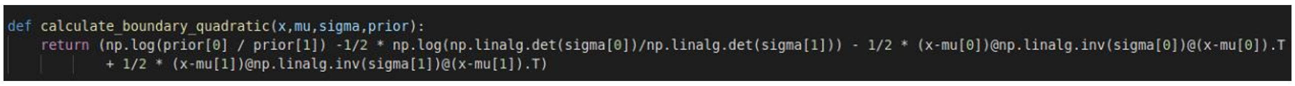
QUESTION 3:

Quadratic and linear classifiers implemented by directly computing discriminant functions. You can see the function that compute Quadratic Discriminant Classifier. Note that these function only calculates scores, then by using numpy highest scored class will be assigned to a point. Although Linear Discriminant implemented with vectorized fashion, Quadratic function implemented without vectorization to show both implementation techniques.



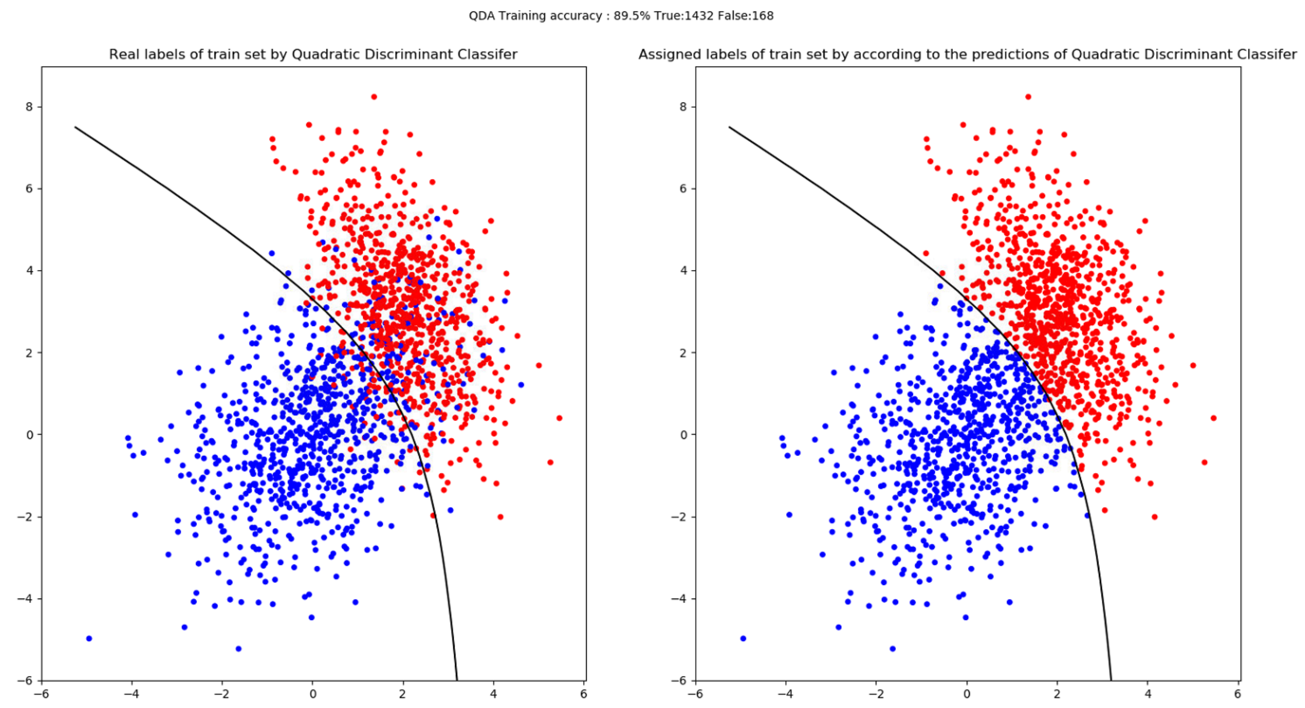
```python
def qda( x, mu, sigmas, prior ):
    num_class = mu.shape[0]
    inv_sigma_list = [np.linalg.inv(sig) for sig in sigmas]
    results = np.zeros((x.shape[0],num_class))
    # Qda applied without vectorization,to show that mods can be done without vectorizing
    for i in range(x.shape[0]):
    # For every class label calculate classification score
        for j in range(num_class):
            term1 = np.float32(-.5*np.log(np.linalg.det(sigmas[j])))
            term2 = -.5*(x[i]-mu[j]).T@inv_sigma_list[j]@(x[i]-mu[j])
            term3 = np.log(prior[j])
            results[i,j] = term1+term2+term3
    return  results
```

Also to find the quadratic boundary, we should compute the points where score values are equal for both classes. Therefore we solve the QDA formula by making equal the scores of two classes and solved together.

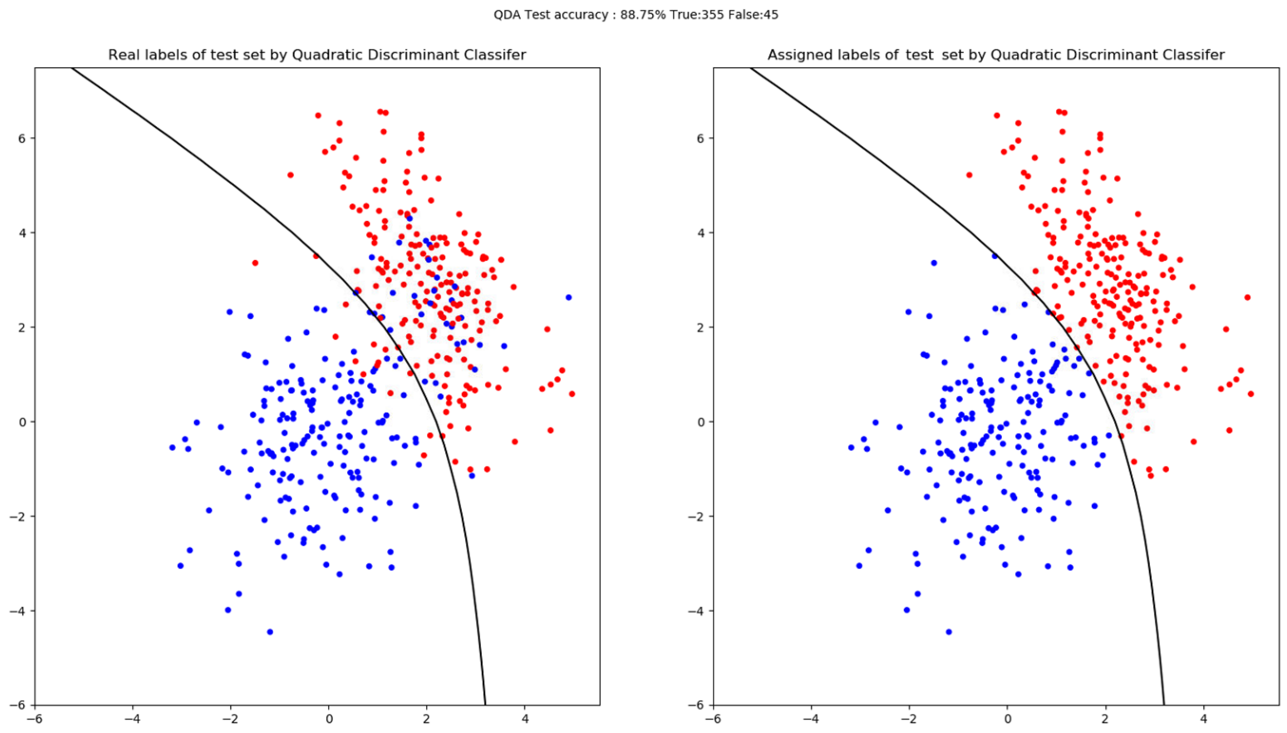


```python
def calculate_boundary_quadratic(x,mu,sigma,prior):
    return (np.log(prior[0] / prior[1]) -1/2 * np.log(np.linalg.det(sigma[0])/np.linalg.det(sigma[1])) - 1/2 * (x-mu[0])@np.linalg.inv(sigma[0])@(x-mu[0]).T
            + 1/2 * (x-mu[1])@np.linalg.inv(sigma[1])@(x-mu[1]).T)
```

Below you can see the results of the training set and test set. Also the accuracies of the each set by giving decision boundary is plotted.

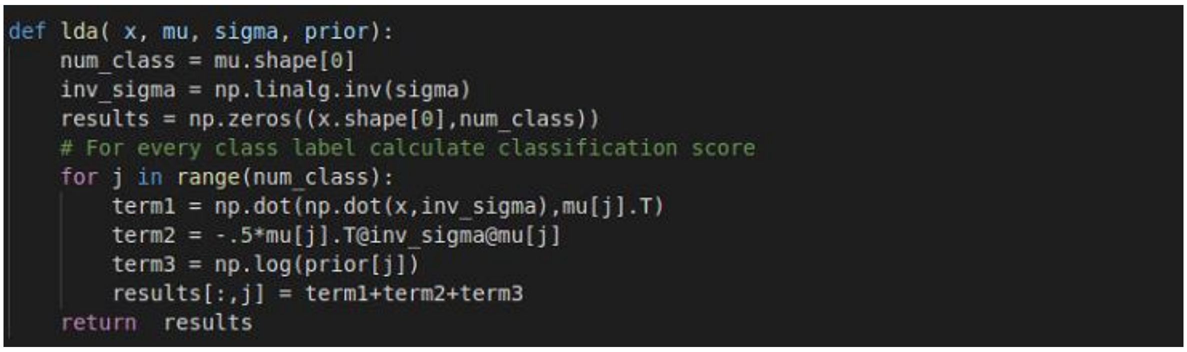


Plots on the left shows the real labels of the set with given decision boundary. Therefore you can see the wrongly specified datapoints on the left. The plot on the right shows our classification results, therefore you can see the class 1 is right part of the decision boundary and class 0 is the left part of the decision boundary. Also below you can see the test set results

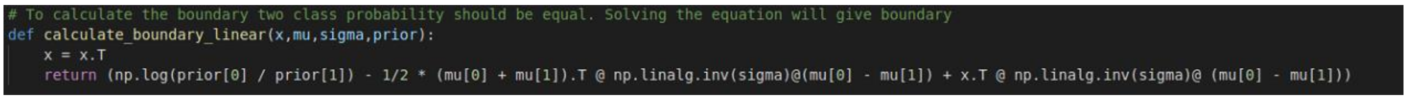


QUESTION 4:

Most of the calculation and plotting part is the same with the Quadratic Discriminant Classifier because QDA is a generalized version of LDA. LDA has an assumption that all the classes have same covariance matrix, where QDA uses distinct covariance matrixes for each classes. In Linear Discriminant Classifier shared covariance matrix is used. You can see the calculation of the LDA score and decision boundary.

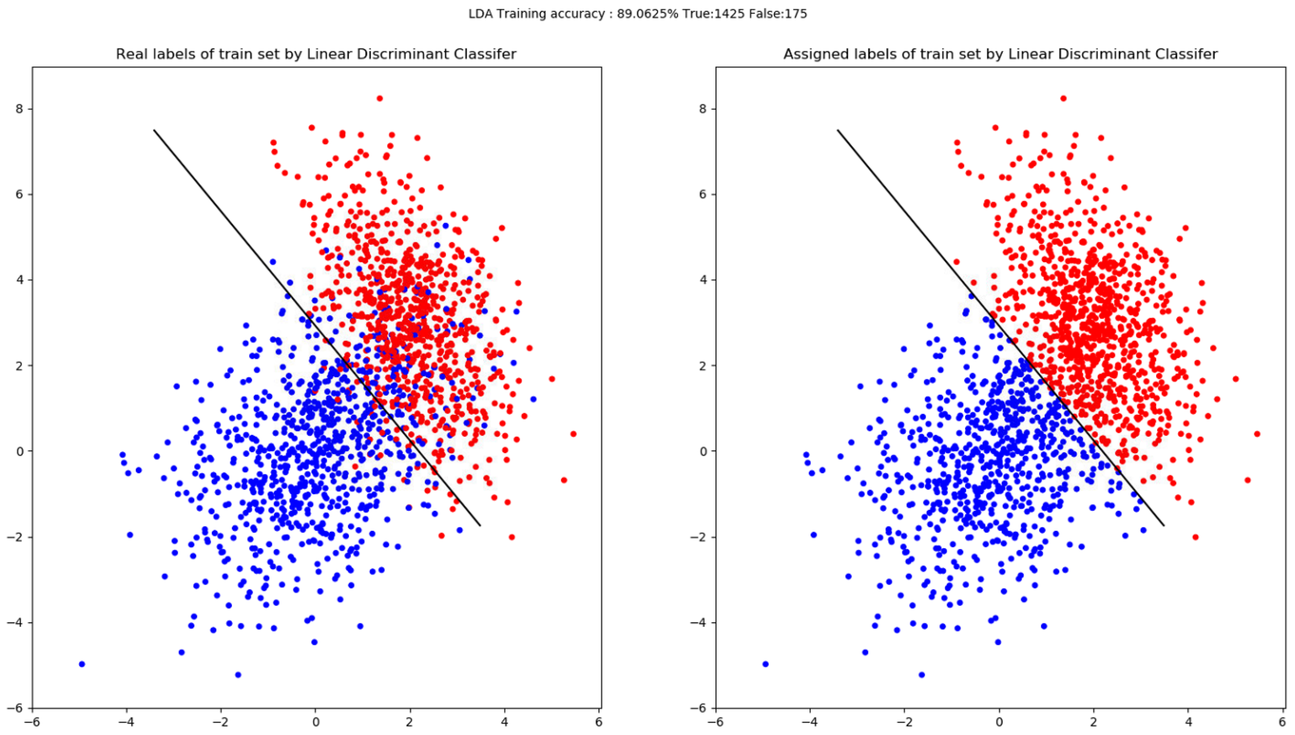


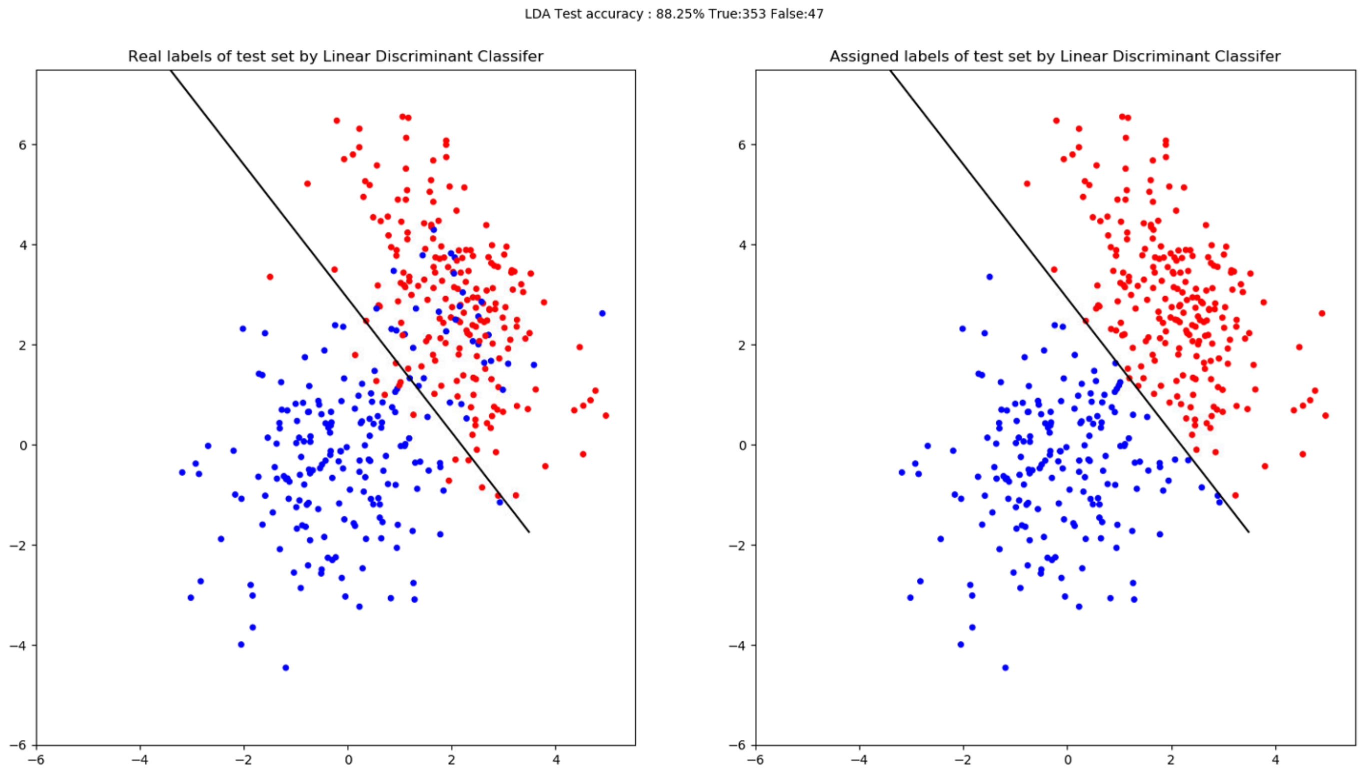
```python
def lda( x, mu, sigma, prior):
    num_class = mu.shape[0]
    inv_sigma = np.linalg.inv(sigma)
    results = np.zeros((x.shape[0],num_class))
    # For every class label calculate classification score
    for j in range(num_class):
        term1 = np.dot(np.dot(x,inv_sigma),mu[j].T)
        term2 = -.5*mu[j].T@inv_sigma@mu[j]
        term3 = np.log(prior[j])
        results[:,j] = term1+term2+term3
    return  results
```



```python
# To calculate the boundary two class probability should be equal. Solving the equation will give boundary
def calculate_boundary_linear(x,mu,sigma,prior):
    x = x.T
    return (np.log(prior[0] / prior[1]) - 1/2 * (mu[0] + mu[1]).T @ np.linalg.inv(sigma)@(mu[0] - mu[1]) + x.T @ np.linalg.inv(sigma)@ (mu[0] - mu[1]))
```

Note that "@" is used for matrix multiplication and this is a vectorized version of the lda and therefore it may be looking different than the mathematical formula but it gives same results. Modifications are only made for the vectorization of the formula, therefore the function above takes [N,feature] dataset and returns [N,num_class] matrix where each datapoint has class scores for each class. Again label assigning process is made by numpy argmax to set the label with maximum class score .





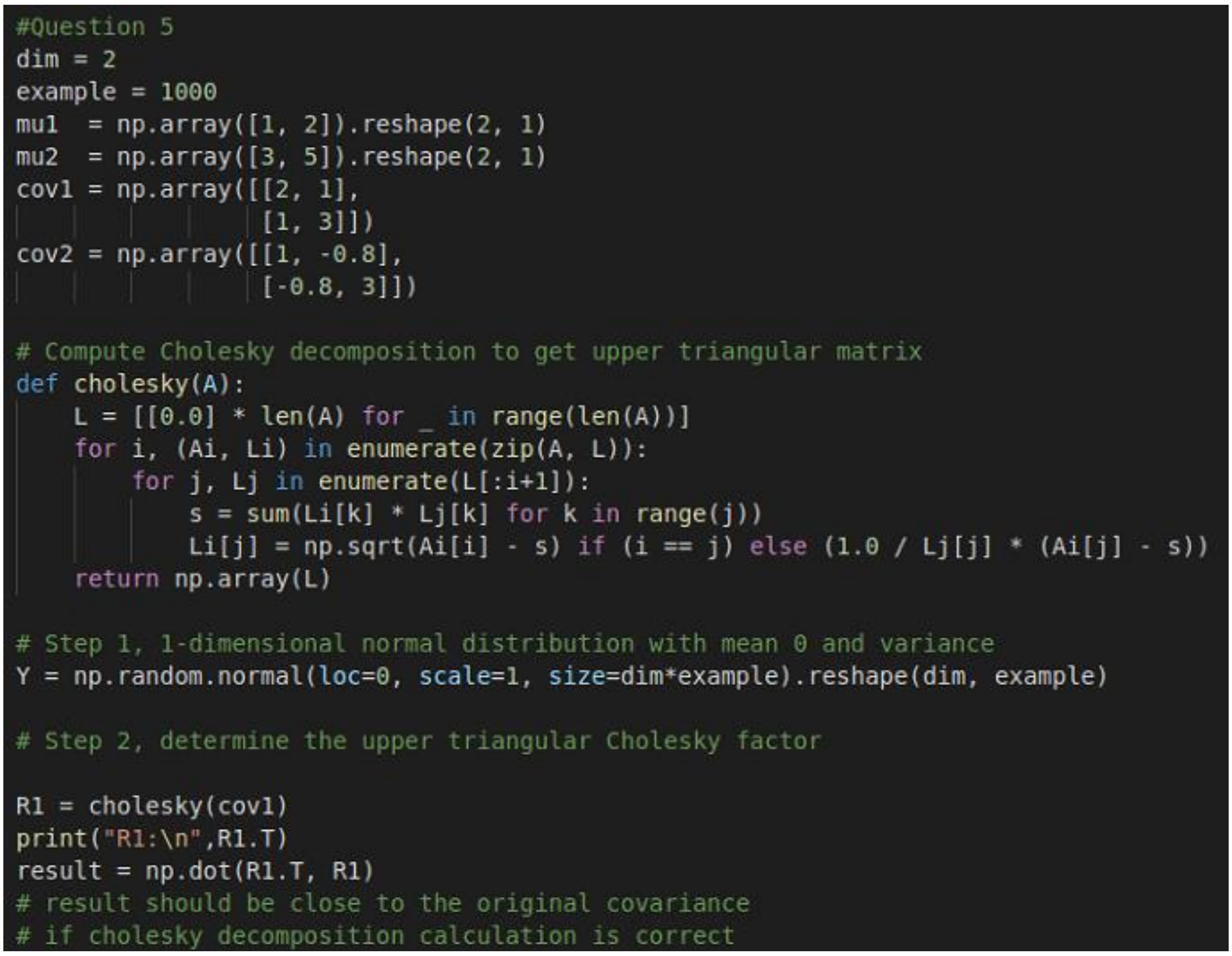
QUESTION 5:

3 Steps is necessary to generate Gaussian dataset.

1- create an MxN vector Y, each of whose elements is a sample of the 1-dimensional normal distribution with mean 0 and variance 1;
2- determine the upper triangular Cholesky factor R of the matrix A, so that A = R' * R;
3- compute X = MU + R' * Y

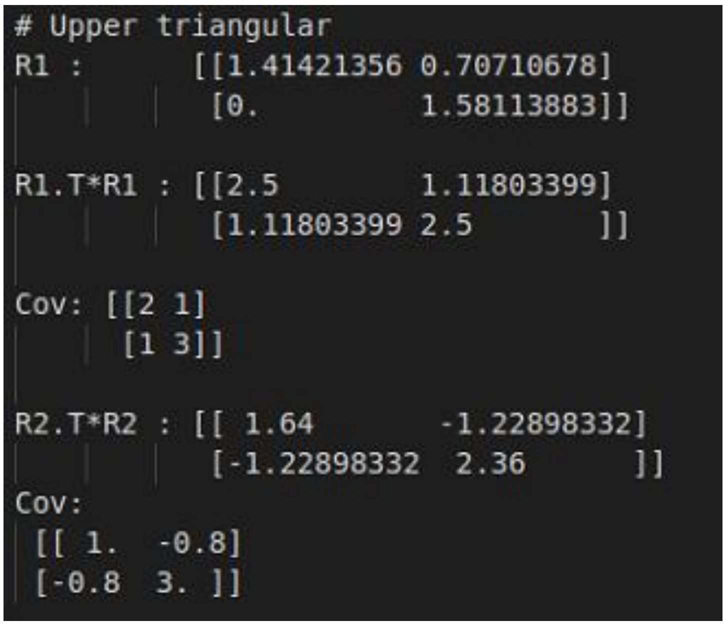
First step is easy if you use numpy. By using numpy.random.normal we sample from normal distribution with mean 0 and variance 1

Then for step 2 we need to do Cholesky decomposition. Code below shows how to calculate upper triangular Cholesky factor.



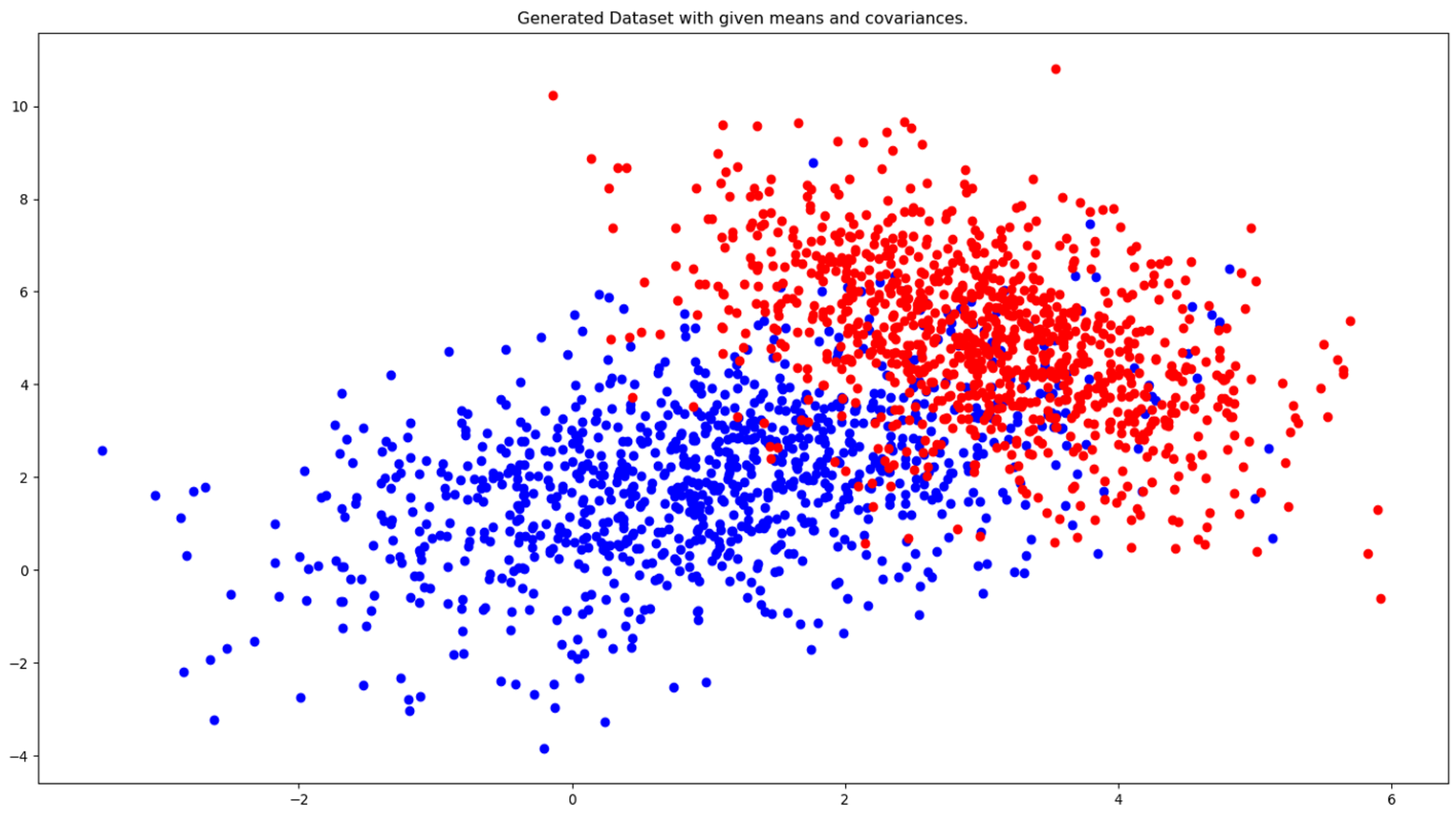
```
#Question 5
dim = 2
example = 1000
mu1  = np.array([1, 2]).reshape(2, 1)
mu2  = np.array([3, 5]).reshape(2, 1)
cov1 = np.array([[2, 1],
                 [1, 3]])
cov2 = np.array([[1, -0.8],
                 [-0.8, 3]])

# Compute Cholesky decomposition to get upper triangular matrix
def cholesky(A):
    L = [[0.0] * len(A) for _ in range(len(A))]
    for i, (Ai, Li) in enumerate(zip(A, L)):
        for j, Lj in enumerate(L[:i+1]):
            s = sum(Li[k] * Lj[k] for k in range(j))
            Li[j] = np.sqrt(Ai[i] - s) if (i == j) else (1.0 / Lj[j] * (Ai[j] - s))
    return np.array(L)

# Step 1, 1-dimensional normal distribution with mean 0 and variance
Y = np.random.normal(loc=0, scale=1, size=dim*example).reshape(dim, example)

# Step 2, determine the upper triangular Cholesky factor

R1 = cholesky(cov1)
print("R1:\n",R1.T)
result = np.dot(R1.T, R1)
# result should be close to the original covariance
# if cholesky decomposition calculation is correct
```

If we have calculated Cholesky factor matrix we should be able to get the original covariance matrix. Here are the results of the script fo both cov1 and cov2.



```
# Upper triangular
R1 :      [[1.41421356 0.70710678]
           [0.         1.58113883]]

R1.T*R1 : [[2.5        1.11803399]
           [1.11803399 2.5       ]]

Cov: [[2 1]
      [1 3]]

R2.T*R2 : [[ 1.64        -1.22898332]
           [-1.22898332  2.36      ]]
Cov:
 [[ 1.  -0.8]
 [-0.8  3. ]]
```

Then as a last step we should calculate X = MU + R' * Y where X contains our dataset. Doing this for two classes with different mu and covariance values we can create the wanted dataset. You can see the result of the generated dataset with two different classes where each of them contains 1000 examples.



From the covariance matrix and mean vector we could predict the dataset distribution and result of the computation is same with our prediction.