Pattern Recognition Assignment No:2
Code
18MCMI09
Krishnan.S

All the algorithms were implemented using Python3. The libraries that were used were Numpy, Matplotlib and scikit-learn. Outputs aren't shown in this file because all of those plots needed to be shown in the analysis file. There aren't any other outputs except for the plot.

1.
The PCA function takes the data matrix D as input. The LDA function takes the data matrix for two classes as input. So you have to split D into two classes, D1 & D2 and then give it to the function.
np.mean() returns the mean of the matrix or vector given along a specified axis. **np.matmul()** returns the product of two matrices. **np.linalg.eig()** returns the eigenvalues and eigenvectors of a given square matrix. **np.dot()** returns the inner product of two vectors or matrices.

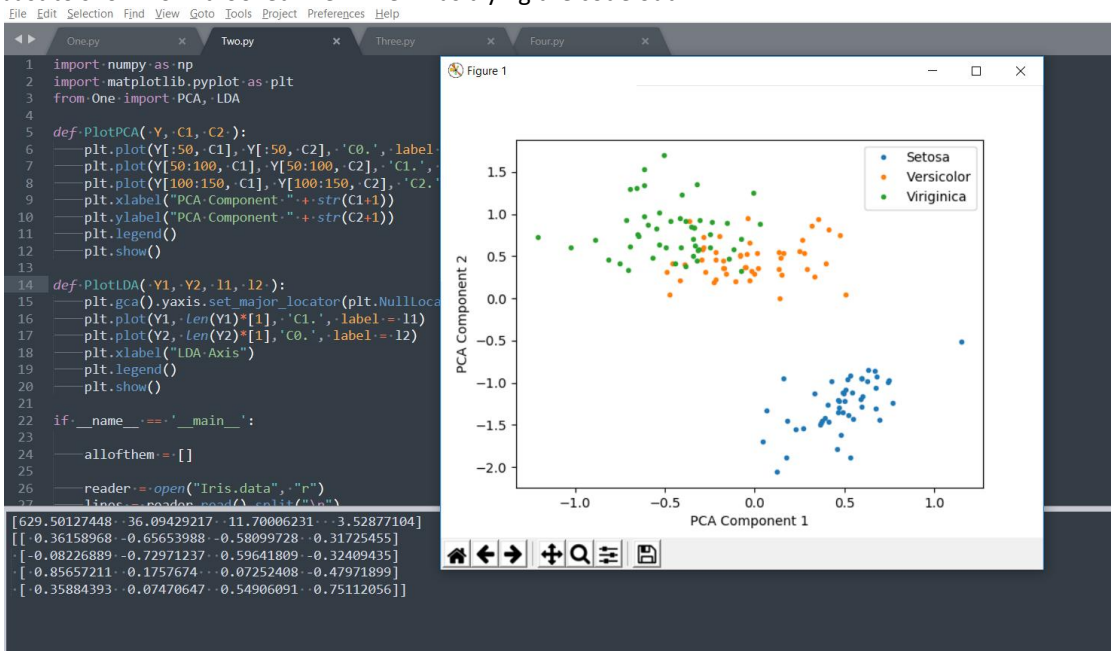
Code for PCA:
```
def PCA( D ):
    mu = np.mean(D, axis = 0)
    X = D - mu
    S = np.matmul(X.T,X)
    EigenValues, EigenVectors = np.linalg.eig(S)

    EigenValues = np.array(EigenValues)
    EigenVectors = np.array(EigenVectors)

    Y = np.dot(X, EigenVectors.T)

    return Y, EigenValues, EigenVectors
```

Code for LDA:
```
def LDA( D1, D2 ):
    mu1 = np.mean(D1, axis = 0)
    mu2 = np.mean(D2, axis = 0)

    S1 = np.matmul((D1-mu1).T, (D1-mu1))
    S2 = np.matmul((D2-mu2).T, (D2-mu2))

    SW = S1 + S2
    muD = np.atleast_2d(mu1-mu2)
    SB = np.matmul(muD.T, muD)

    SWInv = np.linalg.pinv(SW)
    A = np.matmul(SWInv, SB)
    EigenValues, _ = np.linalg.eig(A)

    V = np.matmul(SWInv, muD.T)

    Y1 = np.dot(D1, V)
    Y2 = np.dot(D2, V)
    return Y1, Y2
```

2.
The classes were explicity stated in the PlotPCA and PlotLDA points for simplicity. The PlotPCA takes the projected data matrix as input along with the components you want to display (C1 and C2). If C1 and C2 are 0 and 1 respectively, it displays the first two PCA components. The PlotLDA function takes

the projected data matrix for each class (Y1 and Y2) and the labels for the classes (which are either Setosa, Versicolor or Virginica).
The data was read from the Iris.data file and then stored in a list called “allofthem”. PCA is called on this list and then the points are plotted for the components 1vs2, 1vs3 and 2vs3. The first 50 of the dataset belong to Setosa, the next 50 to Versicolor and the final 50 to Viriginica. So the list was split into these three parts and then fed to the LDA function two at a time. The output plots are shown in the analysis file.

Just to show how it looked like while I was trying the code out



Code:

```
def PlotPCA( Y, C1, C2 ):
        plt.plot(Y[:50, C1], Y[:50, C2], 'C0.', label = "Setosa")
        plt.plot(Y[50:100, C1], Y[50:100, C2], 'C1.', label = "Versicolor")
        plt.plot(Y[100:150, C1], Y[100:150, C2], 'C2.', label = "Viriginica")
        plt.xlabel("PCA Component " + str(C1+1))
        plt.ylabel("PCA Component " + str(C2+1))
        plt.legend()
        plt.show()

def PlotLDA( Y1, Y2, l1, l2 ):
        plt.gca().yaxis.set_major_locator(plt.NullLocator())
        plt.plot(Y1, len(Y1)*[1], 'C1.', label = l1)
        plt.plot(Y2, len(Y2)*[1],'C0.', label = l2)
        plt.xlabel("LDA Axis")
        plt.legend()
        plt.show()

if __name__ == '__main__':
        allofthem = []
        reader = open("Iris.data", "r")
        lines = reader.read().split("\n")
        for line in lines:
                line = line.split(",")
                allofthem.append(list(map(float, line[:-1])))

        allofthem = np.array(allofthem)

        Y, Eval, Evec = PCA( allofthem )
        print(Eval)
```

```
    print(Evec)

    PlotPCA( Y, 0, 1)
    PlotPCA( Y, 0, 2)
    PlotPCA( Y, 1, 2)

    Y1, Y2 = LDA(allofthem[0:50],allofthem[50:100])
    PlotLDA( Y1, Y2, "Setosa", "Versicolor" )
    Y1, Y2 = LDA(allofthem[0:50],allofthem[100:150])
    PlotLDA( Y1, Y2, "Setosa", "Viriginica" )
    Y1, Y2 = LDA(allofthem[50:100],allofthem[100:150])
    PlotLDA( Y1, Y2, "Versicolor", "Viriginica" )
```

3.

The plotting functions for PCA and LDA are identical to the ones above. I explicitly gave the classes for the PlotPCA function for clear labelling. The ScreePlot function takes a vector of eigenvalues and plots a bar graph from those values. The dataset and the classes are different from the last question. The first 44 of this dataset contain healthy users, the rest have cancer. So the list was split at 44 and these two parts were fed to the LDA function. This uses the UCI Arcene Cancer dataset. Running this program will take a long time on a cpu. It took me around 15 minutes for PCA and another 30 minutes to run LDA since the dataset is so large.

The output plots are all present in the analysis file.

Code:

```
def PlotPCA( Y, C1, C2 ):
    plt.plot(Y[:4, C1], Y[0:4, C2], 'C0.', label = "Healthy")
    plt.plot(Y[44:50, C1], Y[44:50, C2], 'C1.', label = "Has Cancer")
    plt.xlabel("PCA Component " + str(C1+1))
    plt.ylabel("PCA Component " + str(C2+1))
    plt.legend()
    plt.show()

def MakeScree( EVals ):
    plt.bar(np.arange(len(EVals)), EVals)
    plt.show()

def PlotLDA( Y1, Y2, l1, l2 ):
    plt.gca().yaxis.set_major_locator(plt.NullLocator())
    plt.plot(Y1, len(Y1)*[1], 'C1.', label = l1)
    plt.plot(Y2, len(Y2)*[1],'C0.', label = l2)
    plt.xlabel("LDA Axis")
    plt.legend()
    plt.show()

if __name__ == '__main__':
    #In the UCI Arcene training set, the first 44 are healthy and the rest are not healthy
    allofthem = []
    reader = open("arcene_train.data", "r")
    lines = reader.read().split("\n")
    lines.pop()
    print(len(lines))
    for line in lines:
        line = line.split()
        allofthem.append(list(map(float, line[:-1])))

    Y, EVals, EVecs = PCA( allofthem )
    PlotPCA( Y, 0, 1 )

    MakeScree( EVals )
```

```
    Y1, Y2 = LDA(allofthem[0:44], allofthem[44:])
    PlotLDA( Y1[:4], Y2[:5], "Healthy", "Has Cancer")
```

4.

The swiss roll dataset was generated using the **make_swiss_roll()** function from the scikit-learn dataset module. It takes the number of points you want to generate as it's input, along with the other parameters to determine the distribution if necessary. kPCA was done using the **KernelPCA()** function from the scikit-learn decomposition module. It takes the number of components wanted and the kernel as an input and returns a Kernel PCA object. Calling the fit_transform() function on the dataset produces the projected data matrix.

LLE was done using the **LocallyLinearEmbedding()** function from the scikit-learn manifold module. It takes the number of components wanted and the number of nearest neighbours (k) needed for local approximation as its input and returns an LLE object. Again, calling the fit_transform() function on the dataset produces the projected data matrix.

The output plots are all present in the analysis file.

Code:

```
def PlotComps( Y, L1, L2 ):
    plt.plot(Y[:,0], Y[:,1], 'C0.')
    plt.xlabel(L1)
    plt.ylabel(L2)
    plt.legend()
    plt.show()

if __name__ == '__main__':
    X, _ = make_swiss_roll(1000)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(X[:,0], X[:,1], X[:, 2])
    plt.show()

    T = KernelPCA(n_components = 2, kernel = "linear")
    YKpca = T.fit_transform(X)

    T = KernelPCA(n_components = 2, kernel = "rbf")
    YKpca2 = T.fit_transform(X)

    LLE = LocallyLinearEmbedding(n_components = 2, n_neighbors = 5)
    YLLE = LLE.fit_transform(X)

    LLE = LocallyLinearEmbedding(n_components = 2, n_neighbors = 150)
    YLLE2 = LLE.fit_transform(X)

    Y, _, _ = PCA(X)

    PlotComps( Y, "PCA Component 1", "PCA Component 2")
    PlotComps( YKpca, "kPCA Component 1", "kPCA Component 2")
    PlotComps( YKpca2, "kPCA Component 1", "kPCA Component 2")
    PlotComps( YLLE, "LLE Component 1", "LLE Component 2")
    PlotComps( YLLE2, "LLE Component 1", "LLE Component 2")


    #Creating punctures in the dataset
    X = np.concatenate( (X[0:120], X[150:240], X[300:500], X[500:540], X[600:900], X[950:]) )
    ax.scatter(X[:,0], X[:,1], X[:, 2], cmap = plt.get_cmap('Oranges'))
    plt.show()

    T = KernelPCA(n_components = 2, kernel = "linear")
    YKpca = T.fit_transform(X)
```

```
T = KernelPCA(n_components = 2, kernel = "rbf")
YKpca2 = T.fit_transform(X)

LLE = LocallyLinearEmbedding(n_components = 2, n_neighbors = 5)
YLLE = LLE.fit_transform(X)

Y, _, _ = PCA(X)

PlotComps( Y, "PCA Component 1", "PCA Component 2")
PlotComps( YKpca, "kPCA Component 1", "kPCA Component 2")
PlotComps( YKpca2, "kPCA Component 1", "kPCA Component 2")
PlotComps( YLLE, "LLE Component 1", "LLE Component 2")
```
