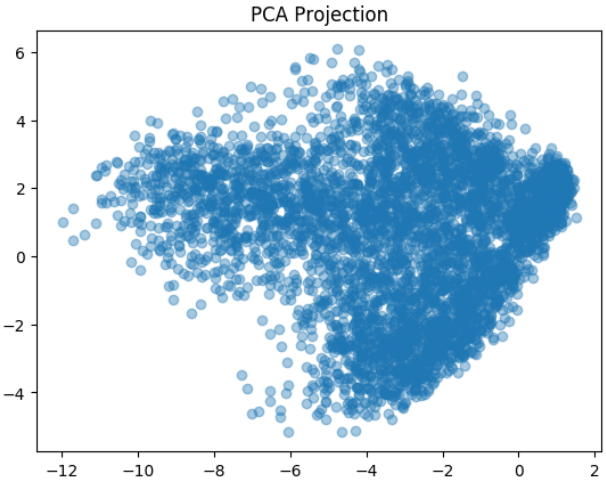
0780804 – 高黃江
Hoang-Giang, Cao

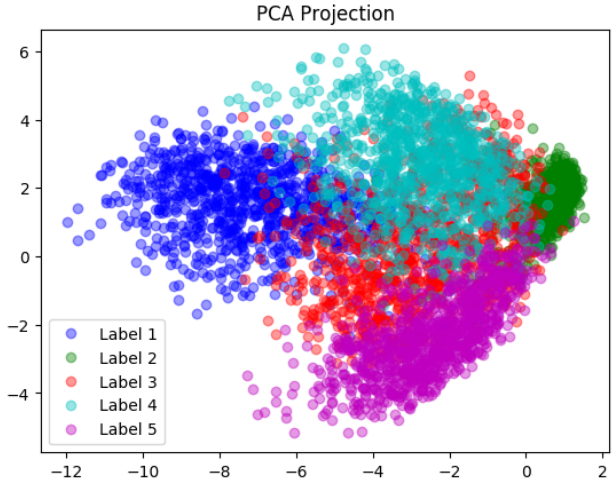
HOME WORK 7
MACHINE LEARNING
PCA LDA EIGENFACES

1. PCA Projection

a) No Clustering

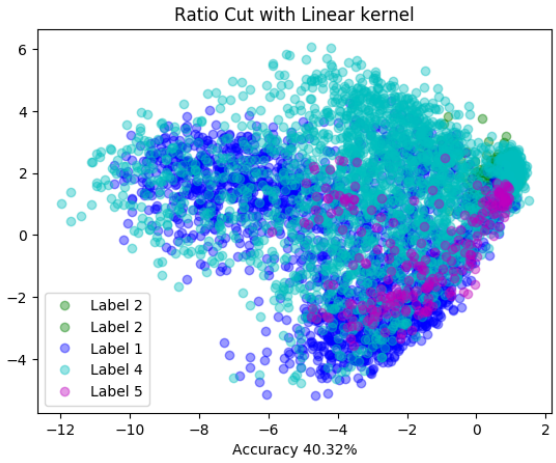


No coloring

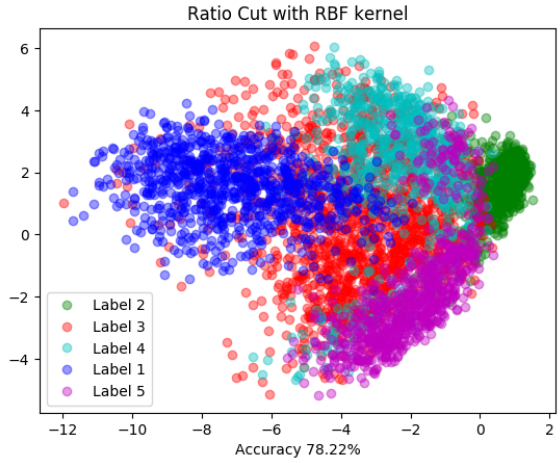


Coloring by T_Train (ground truth)

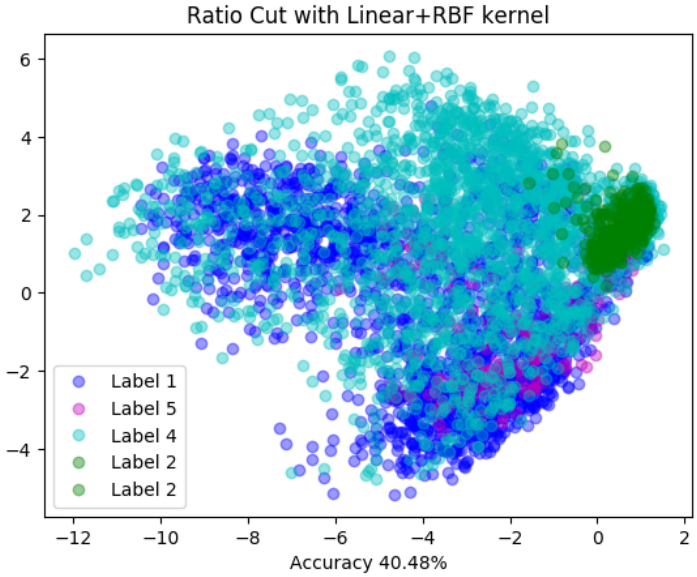
b) Ratio Cut



Ratio Cut with Linear Kernel

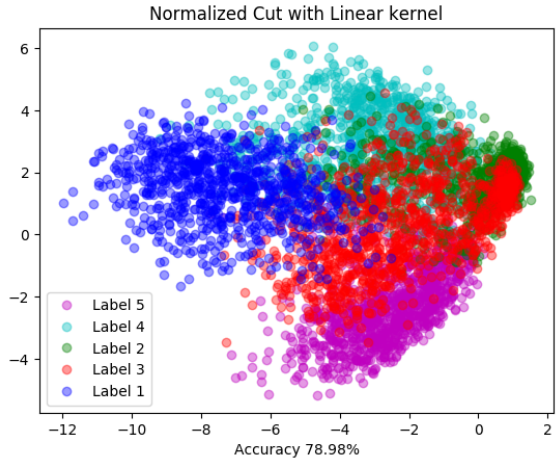


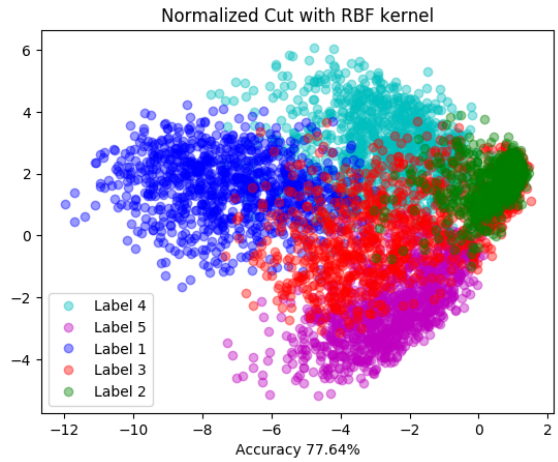
Ratio Cut with RBF kernel sigma = 0.285



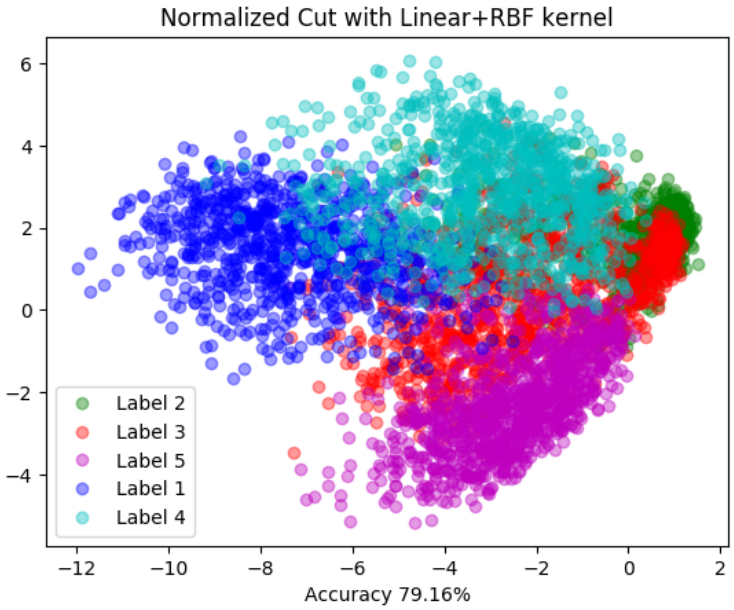
Ratio Cut with Linear + RBF kernel sigma = 5

c) Normalized Cut





Normalized Cut with Linear Kernel Normalized Cut with RBF kernel sigma = 9

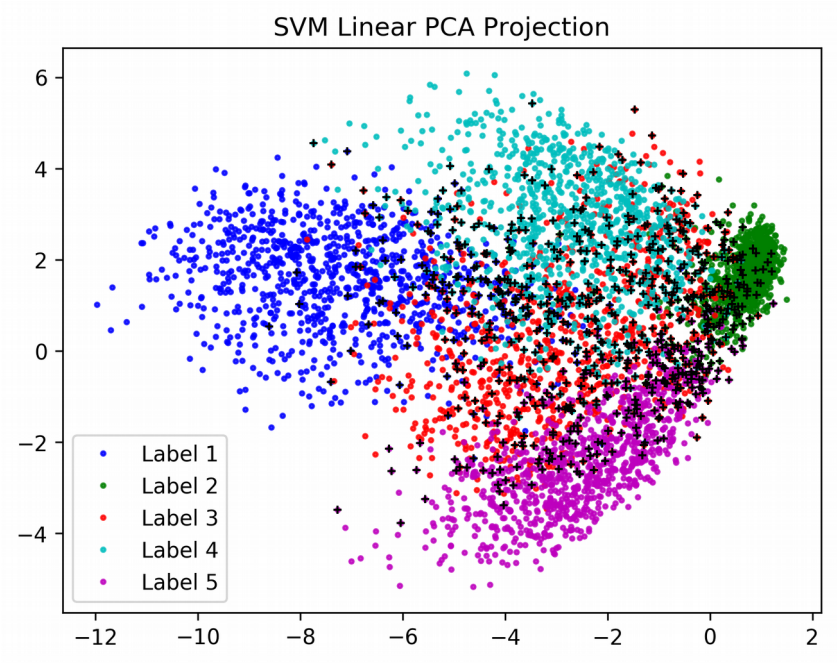


Normalized Cut with Linear + RBF kernel sigma = 7

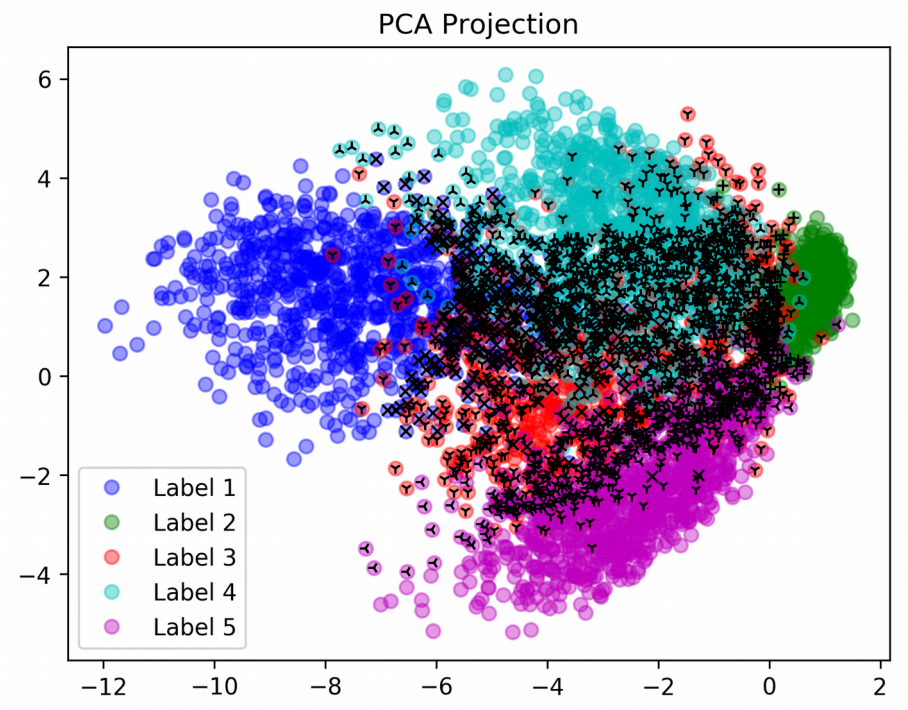
- Ground truth showed above.
- Points that have the same label are painted by the same color among the figures
- The initialization for K-mean is picking central points for each cluster randomly

2. Visualize Support Vector

a) Linear Kernel

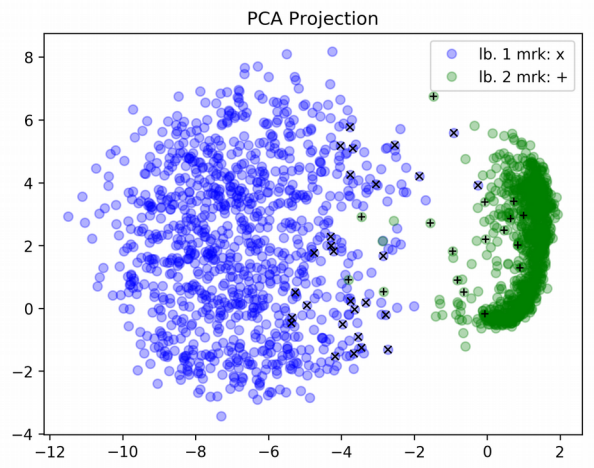


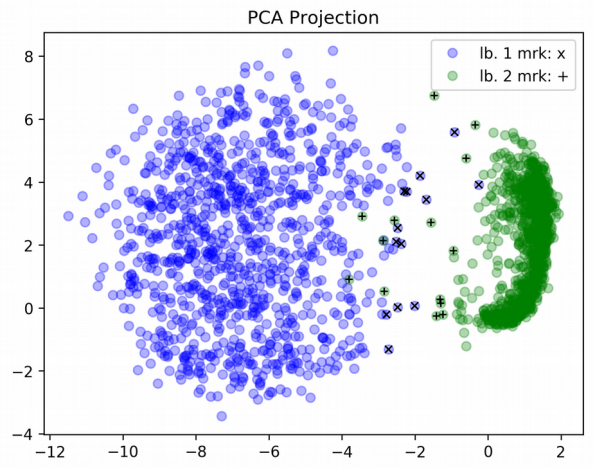
Visualize Support Vector from SVM, using Linear Kernel (707 support vectors)



Support vetor obtain by using 2D data PCA projected by PCA

- I also do SVM based on 2D data that is projected by PCA. We can see that the support vector is about make the boundaries around their clusters.
- To make it clearly, I do SVM and show the support vector using 2 clusters only
- We can see that using 2D data to do SVM, the data is separated by “2D” way, while using the original data, the support vector is more fit

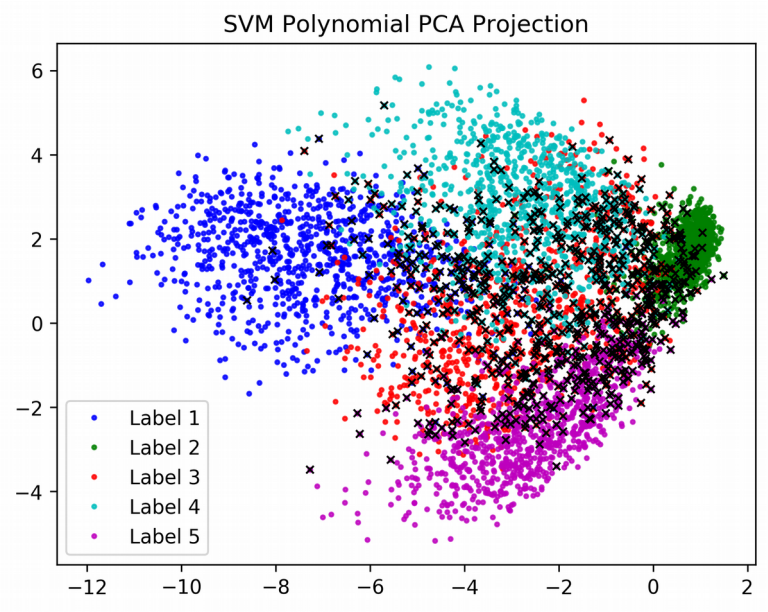




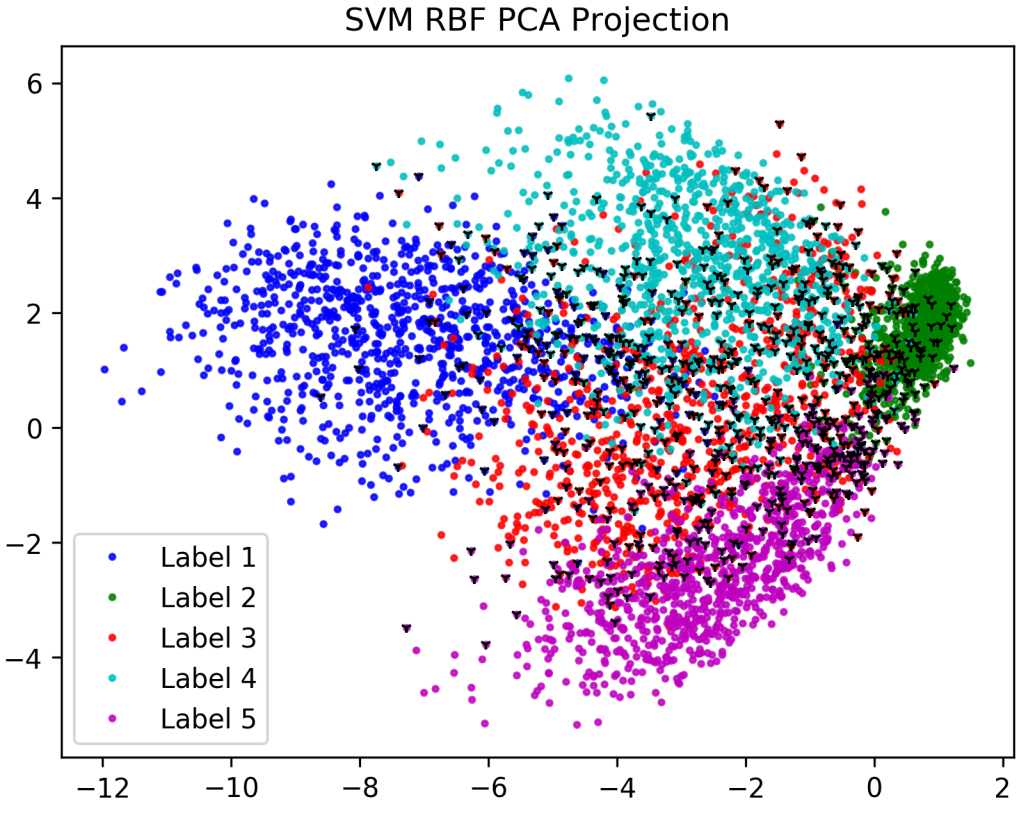
Using original data

Using 2D data projected by PCA

b) Polynomial kernel

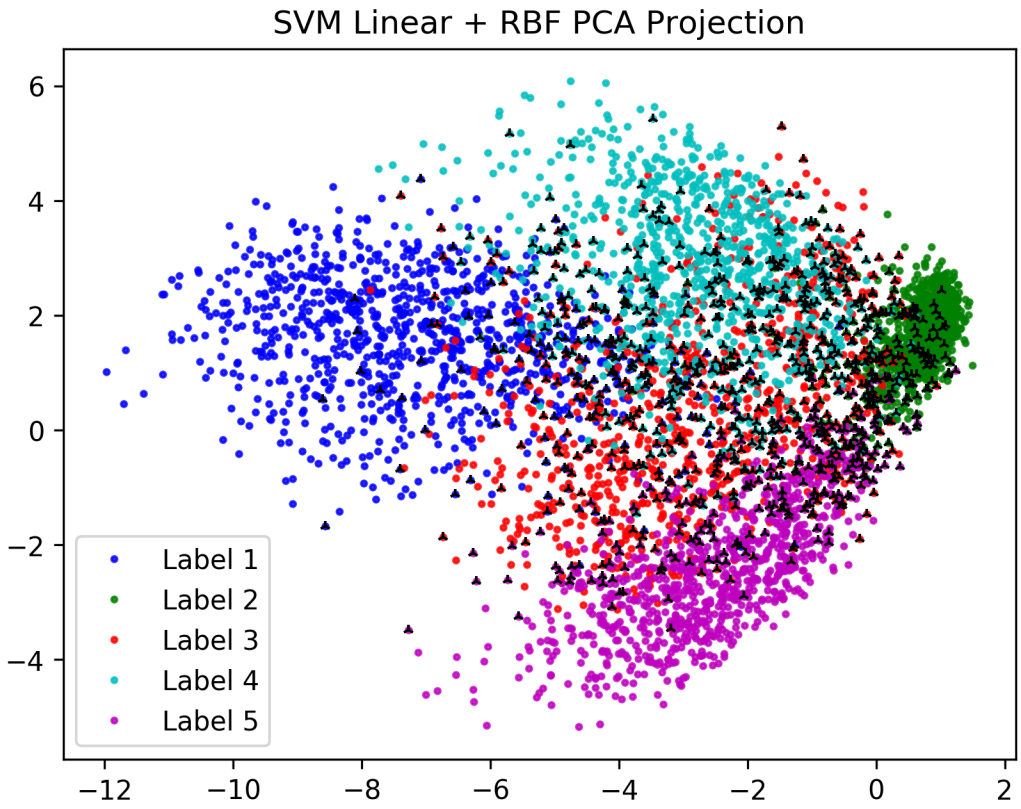


c) RBF with gamma = 0.0625 (obtain from homework #5)



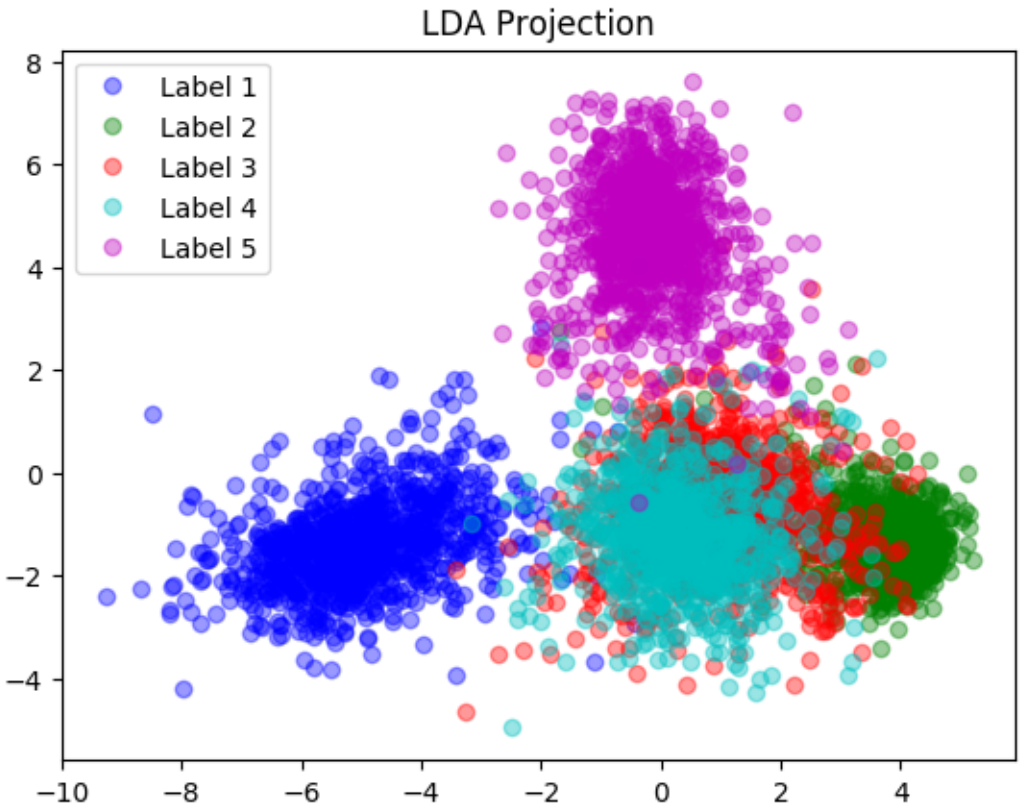
Support vector using RBF kernel (702 support vectors)

d) Linear + RBF kernel



Support vector using Linear + RBF kernel (909 support vectors)

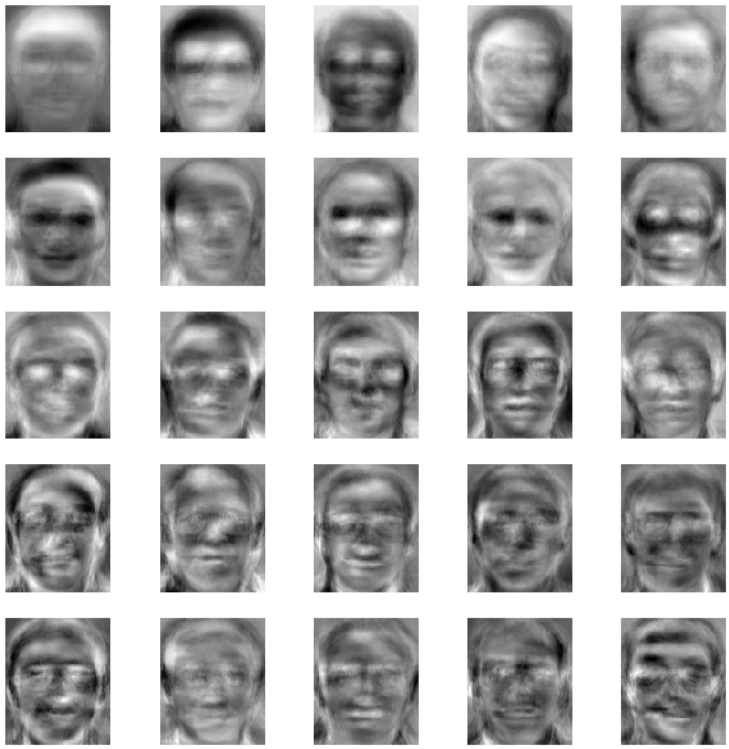
3. **LDA Projection**



LDA Projection. Colored by T_Train

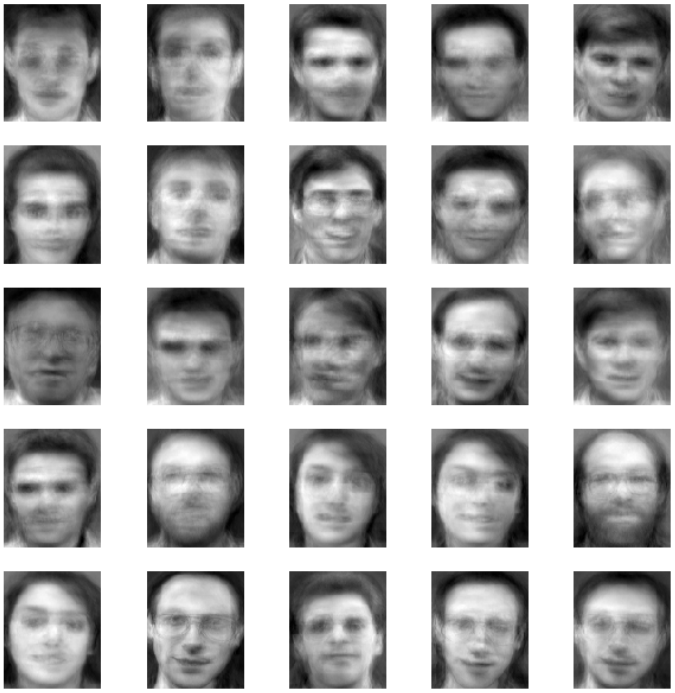
- LDA tries to maximize the Between-Class and within-Class, so that the classes are more seperated.
- While the PCA only try to maximize the variance of the data to keep the most different information from the original data.

4. **Eigenfaces**



The first 25 eigen faces

Original Faces



Reconstructed Faces Using 25 Principal Components

Using 100 Principal components

Using 200 Principal Components

The original faces

Using 400 principal Components

- Calculate the eigen vector of the Matrix M might be time consuming. There is another way to reduce this computation. That is instead of using the fomular M = A.dot(A.T). the dimesion is M^2 x M^2. With A is the centrerdData. and M is the dimeson of the data (119x92).
- We can compute M = A.T.dot(A) . Thereform the M has the size N^2 x N^2, N is the number of data which is much smaller than M. The size of M will be reduces.
- Then to reconstruct the original Eigen vector that have the same size to the original data. We just need to do ReconstructedPCA = A.dot(PCA)
- It is written in PCAFace.py.

CODE EXPLANATION

- PCA

import necessery libraries

```
import csv # for CSV reader
import numpy as np
from math import *
import numpy.matlib
from numpy import linalg as LA
import matplotlib.pyplot as plt
import matplotlib.colors
TrainData = []
nData =0
```

Load data from csv files

```
#TRAIN DATA
with open('X_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                tempRow = []
                for i in range (28*28):
                        tempRow.append((float)(row[i]))
                TrainData.append(tempRow)

TrainData = np.array(TrainData)
nDimesion = TrainData.shape[1]
nData = TrainData.shape[0]

TrainLabel = []
with open('T_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                TrainLabel.append ((float)(row[0]))
TrainLabel = np.array(TrainLabel)
```

Calculate the mean of the data, sum all of the data and divide by number of data

```
mean = np.mean(TrainData,axis=0)
```

Centralize data by subtract all of the data to the mean point

```
centered=TrainData-np.matlib.repmat(mean,nData,1)
```

Calculate the covariance by

```
covariance=(centered.T.dot(centered))/(nData-1)
```

Solving eigen problem of covariance matrix, use 2 eigen vectors that are related 2 largest eigen values. DP = 2

```
DP = 2
eigenValues, eigenVectors = np.linalg.eig(covariance)
largestEigenVectorIndex =  np.argsort(eigenValues.real)
largestEigenVectorIndex = largestEigenVectorIndex[::-1][:DP]
```

Using the first 2 eigen vectors (orders by their eigen values respectively)

```
PCA =  eigenVectors[:,largestEigenVectorIndex]
```

Project data to new space PCA. W = $\bar{X}$.dot(PCA) with $\bar{X}$ is the centralized data

```
W = centered.dot (PCA)
```

Plotting data

```
#plot no color
plt.plot (W[:,0].real,W[:,1].real,marker='o',alpha =0.4,linestyle='none')
plt.title("PCA Projection")
plt.savefig("Report/PCANoColored.png", bbox_inches="tight")
plt.clf()
```

Coloring data by true label.

```
color = ['b','g','r','c','m']
for k in range (5):
        plt.plot (W[k*1000:(k+1)*1000,0],W[k*1000:
(k+1)*1000,1],marker='o',color=color[k],alpha =0.4,linestyle='none', label="Label " +
str(k+1) )
plt.title("PCA Projection")
plt.legend(loc="best")
plt.savefig("Report/PCAColored.png", bbox_inches="tight")
```

Reconstruct to the original data by ReconstructedData = W.dot(PCA.T)

```
reconstructImage = False
if (reconstructImage== True) :
        W=centered.dot(PCA)
        print (W.shape)
        reconstruct  = W.dot(PCA.T) + mean
        for i in range (10) :
                img =  np.reshape (reconstruct[i].real,(28,28))
                plt.imshow(img,cmap='gray')
                plt.show()
                plt.clf()
```

- Ratio Cut

Import necessary libraries

```
import csv # for CSV reader
import numpy as np
from math import *
import sys
import random
import matplotlib.pyplot as plt
from math import exp,sqrt
from matplotlib.lines import Line2D
import time
import numpy.matlib

TrainData = []
TrainLabel = []
nData = 0
```

Function to calculate the Euclide distance

```
def squareDif(A,B) :
    dif =np.linalg.norm(A-B)
    return dif
```

RBF functions. RBF kernel is defined as exp(-||Xi - Xj ||/(2*gamma^2)), γ > 0

```
def RBFunction(g,A,B) :
    dif2 = np.linalg.norm(A-B)
    d = np.exp(-(dif2)/(2*g**2))
    #print (d)
    return d
```

Loading csv data

```
#TRAIN DATA
with open('X_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                tempRow = []
                for i in range (28*28):
                        tempRow.append((float)(row[i]))
                TrainData.append(tempRow)

with open('T_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                TrainLabel.append ((float)(row[0]))

TrainData = np.array(TrainData)
TrainLabel = np.array(TrainLabel)
```

Double check the standard deviation. The standard deviation is the square root of the average of the squared deviations from the mean. std = sqrt(mean(abs(x - x.mean())**2))

```
stdv = np.std(TrainData)
print (stdv)

nData = TrainData.shape[0]
```

get the kernel parameter from commandline. Kernel can be Linear or RBF or Linear + RBF

```
# similarity Grahp
kernel=  sys.argv[1]
```

If using Linear Kernel, calculate Linear kernel, defined as K(Xi , Xj) = Xi.T Xj

```
if (kernel=="Linear" or kernel=="linear" or kernel=="l" or kernel=="L"):
        print ("Using Linear Kernel")
        A = TrainData.dot(TrainData.T)
```

Or ussing RBF kernel, defined as exp(-γ||Xi - Xj || 2), γ > 0 . The simarlarity is defined as RBF functions

```
elif (kernel=="RBF" or kernel=="rbf" or kernel=="r" or kernel=="R" ):
        print ("Using RBF, calculating RBF kernel g = " + str (sys.argv[2]))
        start_time = time.time()
        g =  (float) ( sys.argv[2]) # 0.2851
        A = np.zeros((nData,nData))
        for i in range(0,nData):
            for j in range(i, nData,1):
                A[i,j] = RBFunction(g,TrainData[i], TrainData[j])
```

```
                A[j,i] = A[i,j]
        print("time for calculating RBF : " + str((time.time() - start_time)/60) +" minutes"
)
else :
```

Or using Linear +RBF to calculate the similarity among the data points.

```
        print ("Using Linear+RBF Kernel")
        print ("calculating Linear kernel")
        A = TrainData.dot(TrainData.T)
        print ("calculating RBF kernel g = " + str (sys.argv[2]))
        start_time = time.time()
        g =  (float) ( sys.argv[2]) # 0.2851
        for i in range(0,nData):
            for j in range(i, nData,1):
                A[i,j] += RBFunction(g,TrainData[i], TrainData[j])
                A[j,i] = A[i,j]

print (A.shape)
```

Calculating the Degree Graph D by sum all of the row of similarity matrix

```
#degree Graph
print ("calculating D")
D = np.zeros((nData,nData))
for i in range(nData):
    D[i,i] = sum(A[i])
```

Calculate Unnormalized Laplacian matrix L = D- A1

```
#Laplacian
L = D -  A

print ("calculating eigenValues, eigenVectors")
start_time = time.time()
```

Solving Eigen Value and Eigen value of matrix L

```
eigenValues, eigenVectors = np.linalg.eig(L)
#the first K value
```

Sort eigen values to take the first K eigen vectors that are related to the first K smallest eigen values. Transform original data to new data space

```
smallestEigenVector =  np.argsort(eigenValues.real)
print("time for calculating: " + str((time.time() - start_time)/60) +" minutes" )
#5 class
K = 5
print (smallestEigenVector)
newSpace =np.zeros((nData,0))

for i in range(0,K,1):
        newCoor = (np.array(eigenVectors[:,smallestEigenVector[i]]))
        newCoor = np.reshape(newCoor,(nData,1))
        newSpace = np.concatenate((newSpace, newCoor), axis=1)

print (newSpace.shape)
```

Do K-mean on the new data space. Random choose K central points as the mean point of each cluster

```
#random k means
miu =  newSpace [random.sample(range(0, nData), K)]
```

Do K mean 100 times, or ultil the mean of each cluster do not change any more

```
# kmeans
iteration = 0
for it in range (100) :
    #reset for the next iteration
    print ("iteration ",(it+1))
    clusterOriginalData = [[[None] for i in range(0)] for j in range(K)]
    labelOriginalData = [[[None] for i in range(0)] for j in range(K)]
    clusterNewSpaceData = [[[None] for i in range(0)] for j in range(K)]
```

Do K-mean on each data, cluster data to the nearest central point.

```
    for i in range (nData) :
        minD = 999999
        selectedK = - 1
        for k in range (K) :
                tempD = sqrt(squareDif( newSpace[i] , miu[k]))
```

```
                if (tempD < minD) :
                    minD = tempD
                    selectedK = k

        clusterOriginalData[selectedK].append (TrainData[i])
        labelOriginalData[selectedK].append (TrainLabel[i])
        clusterNewSpaceData[selectedK].append (newSpace[i])
```

Calculate new mean, compare to the current mean, if the mean do not change, return the clustering result. Or else, update the current mean

```
    newMean = np.zeros((K,K))
    for k in range (K) :
        newMean[k] = np.mean(clusterNewSpaceData[k],axis =0)
    if ( sqrt(squareDif(newMean, miu)) < 0.00001):
        iteration = it
        break
    miu = newMean.copy()


print ("converged after ", iteration, " interation(s)")
```

Calculate the accuracy. Label of a cluster is defined as the true label that appear most in the cluster #trueLabel. Therefore, the accuracy of each cluster is calculated by dividing #trueLabel to the number data of the cluster.

```
totalAccuracy = 0
mappingLabel = np.zeros(K)
for k in range (K) :
        (values,counts) = np.unique (labelOriginalData[k], return_counts=True)
        ind = np.argmax(counts)
        labeled = values[ind]
        mappingLabel[k] = labeled
        print ("Label: " + str(labeled))
        accuracy = counts[ind]
        totalAccuracy+=accuracy
        print ("accuracy: ", (accuracy*100.0/len(labelOriginalData[k])), "%   "  ,str
(accuracy) +"/" + str(len(labelOriginalData[k])) )
print ("Total Accuracy: ", (totalAccuracy*100.0/nData), "%" )
```

Project data using PCA. The same as doing PCA above

```
#PCA PROJECTION
mean = np.mean(TrainData,axis=0)
centered=TrainData-np.matlib.repmat(mean,nData,1)
covariance=(centered.T.dot(centered))/(nData-1)
DP = 2
eigenValues, eigenVectors = np.linalg.eig(covariance)
largestEigenVectorIndex =  np.argsort(eigenValues.real)
largestEigenVectorIndex = largestEigenVectorIndex[::-1][:DP]
PCA =  eigenVectors[:,largestEigenVectorIndex]
color = ['b','g','r','c','m']
#VISUALIZE LATER
for k in range (K) :
        cluster = np.array(clusterOriginalData[k])
        print (cluster.shape[0])
        W = cluster.dot(PCA)
        plt.plot (W[:,0],W[:,1],marker='o',color=color[int (mappingLabel[k])-1],alpha
=0.4,linestyle='none', label="Label " + str(int (mappingLabel[k])) )
plt.title("Ratio Cut with "+ kernel + " kernel")
plt.xlabel("Accuracy " +  str(np.round((totalAccuracy*100.0/nData),2))+ "%" )
plt.legend(loc='best')
plt.savefig("Report/Ratio_"+kernel+".png", bbox_inches="tight")
#plt.show ()
```

- Normalized Cut. It is almost the same to Ratio Cut, except the calcualtion of Nomalized matrix L

Import necessary libraries

```
import csv # for CSV reader
import numpy as np
from math import *
import sys
import random
import matplotlib.pyplot as plt
from math import exp,sqrt
from matplotlib.lines import Line2D
import time
import numpy.matlib

TrainData = []
TrainLabel = []
nData = 0
```

Function to calculate the Euclide distance

```
def squareDif(A,B) :
    dif =np.linalg.norm(A-B)
    return dif
```

RBF functions. RBF kernel is defined as exp(-||Xi - Xj ||/(2*gamma^2)), γ > 0

```
def RBFunction(g,A,B) :
    dif2 = np.linalg.norm(A-B)
    d = np.exp(-(dif2)/(2*g**2))
    #print (d)
    return d
```

Loading csv data

```
#TRAIN DATA
with open('X_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                tempRow = []
                for i in range (28*28):
                        tempRow.append((float)(row[i]))
                TrainData.append(tempRow)

with open('T_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                TrainLabel.append ((float)(row[0]))

TrainData = np.array(TrainData)
TrainLabel = np.array(TrainLabel)
```

Double check the standard deviation. The standard deviation is the square root of the average of the squared deviations from the mean. std = sqrt(mean(abs(x - x.mean())**2))

```
stdv = np.std(TrainData)
print (stdv)

nData = TrainData.shape[0]
```

get the kernel parameter from commandline. Kernel can be Linear or RBF or Linear + RBF

```
# similarity Grahp
kernel=  sys.argv[1]
```

If using Linear Kernel, calculate Linear kernel, defined as K(Xi , Xj) = Xi.T Xj

```
if (kernel=="Linear" or kernel=="linear" or kernel=="l" or kernel=="L"):
        print ("Using Linear Kernel")
        A = TrainData.dot(TrainData.T)
```

Or ussing RBF kernel, defined as exp(-γ||Xi - Xj || 2), γ > 0 . The simarlarity is defined as RBF functions

```
elif (kernel=="RBF" or kernel=="rbf" or kernel=="r" or kernel=="R" ):
        print ("Using RBF, calculating RBF kernel g = " + str (sys.argv[2]))
        start_time = time.time()
        g =  (float) ( sys.argv[2]) # 0.2851
        A = np.zeros((nData,nData))
        for i in range(0,nData):
            for j in range(i, nData,1):
                A[i,j] = RBFunction(g,TrainData[i], TrainData[j])
                A[j,i] = A[i,j]
        print("time for calculating RBF : " + str((time.time() - start_time)/60) +" minutes"
)
else :
```

Or using Linear +RBF to calculate the similarity among the data points.

```
        print ("Using Linear+RBF Kernel")
        print ("calculating Linear kernel")
        A = TrainData.dot(TrainData.T)
        print ("calculating RBF kernel g = " + str (sys.argv[2]))
        start_time = time.time()
        g =  (float) ( sys.argv[2]) # 0.2851
        for i in range(0,nData):
            for j in range(i, nData,1):
                A[i,j] += RBFunction(g,TrainData[i], TrainData[j])
                A[j,i] = A[i,j]

print (A.shape)
```

Calculating the Degree Graph D by sum all of the row of similarity matrix

```
#degree Graph
print ("calculating D")
D = np.zeros((nData,nData))
for i in range(nData):
    D[i,i] = sum(A[i])
```

Calculating the Degree Graph D by sum all of the row of similarity matrix

```
print ("calculating Normalized L")
```

For the matrix D, we need to add a small value to avoid deviding by zero when doing the power to minus haft calculate Laplaciance sysmetric matrix according to Ng, Jordan, and Weiss (2002)

$$L_{\text{sym}} := D^{-1/2} L D^{-1/2} = I - D^{-1/2} W D^{-1/2}$$

```
D = D + exp(-18)
#Laplacian
I = np.identity(nData)
D_mSquare = D**(-1/2)
preComputeMatrix = D_mSquare.dot(A)
preComputeMatrix = preComputeMatrix.dot(D_mSquare)

#Shi and Malik (2000)
#preComputeMatrix2 = (D**(-1)).dot (A)

L= I - preComputeMatrix
```

Solving Eigen problem of matrix L. Using the first K eigen vectors that are related to the first K smallest eigen values. Form the new data space.

```
print ("calculating eigenValues, eigenVectors")
start_time = time.time()
eigenValues, eigenVectors = np.linalg.eig(L)
#the first K value
smallestEigenVector =  np.argsort(eigenValues.real)
print("time for calculating: " + str((time.time() - start_time)/60) +" minutes" )
K = 5
print ("construct Matrix V")
V =np.zeros((nData,0),dtype=complex)
for i in range(0,K,1):
        newCoor = (np.array(eigenVectors[:,smallestEigenVector[i]]))
        newCoor = np.reshape(newCoor,(nData,1))
        V = np.concatenate((V, newCoor), axis=1)

print (V.shape)
```

Normalized the new data space. Normalizing the make the L2 norm of each axes of new new space equal to 1

```
print ("construct Matrix U Nomalized V")
# normalized V
U = np.zeros((nData,K),dtype=complex)

for i in range (nData):
        normalized = sqrt ( np.sum ( V[i]**2 ).real ).real
        U[i] = V[i]/normalized
```

Do K-mean on the new data space. The same as the K-mean of the Ratio cut

```
#random k means
miu =  U [random.sample(range(0, nData), K)]
```

```
# kmeans
iteration = 0
for it in range (100) :
    #reset for the next iteration
    print ("iteration ",(it+1))
    clusterOriginalData = [[[None] for i in range(0)] for j in range(K)]
    labelOriginalData = [[[None] for i in range(0)] for j in range(K)]
    clusterVData = [[[None] for i in range(0)] for j in range(K)]
```

Clustering the data to the nearest central points

```
    for i in range (nData) :
        minD = 999999
        selectedK = - 1
        for k in range (K) :
                tempD = sqrt(squareDif( U[i] , miu[k]))
                if (tempD < minD) :
                    minD = tempD
                    selectedK = k

        clusterOriginalData[selectedK].append (TrainData[i])
        labelOriginalData[selectedK].append (TrainLabel[i])
        clusterVData[selectedK].append (U[i])

    newMean = np.zeros((K,K))
```

Update the central points and check terminal condition

```
    for k in range (K) :
        newMean[k] = np.mean(clusterVData[k],axis =0)
    if ( sqrt(squareDif(newMean, miu)) < 0.00001):
        iteration = it
        break
    miu = newMean.copy()


print ("converged after ", iteration, " interation(s)")
```

Label the cluster and calculate the accuracy of each cluster

```
totalAccuracy = 0
mappingLabel = np.zeros(K)
for k in range (K) :
        (values,counts) = np.unique (labelOriginalData[k], return_counts=True)
        ind = np.argmax(counts)
        labeled = values[ind]
        mappingLabel[k] = labeled
        print ("Label: " + str(labeled))
        accuracy = counts[ind]
        totalAccuracy+=accuracy
        print ("accuracy: ", (accuracy*100.0/len(labelOriginalData[k])), "%   "  ,str (accuracy) +"/" +
str(len(labelOriginalData[k])) )
print ("Total Accuracy: ", (totalAccuracy*100.0/nData), "%" )
```

Project data using PCA. The same as doing PCA above

```
#PCA PROJECTION
mean = np.mean(TrainData,axis=0)
centered=TrainData-np.matlib.repmat(mean,nData,1)
covariance=(centered.T.dot(centered))/(nData-1)
DP = 2
eigenValues, eigenVectors = np.linalg.eig(covariance)
largestEigenVectorIndex =  np.argsort(eigenValues.real)
largestEigenVectorIndex = largestEigenVectorIndex[::-1][:DP]
PCA =  eigenVectors[:,largestEigenVectorIndex]
color = ['b','g','r','c','m']
#VISUALIZE LATER
for k in range (K) :
        cluster = np.array(clusterOriginalData[k])
        print (cluster.shape[0])
        W = cluster.dot(PCA)
        plt.plot (W[:,0],W[:,1],marker='o',color=color[int (mappingLabel[k])-1],alpha =0.4,linestyle='none',
label="Label " + str(int (mappingLabel[k])) )
plt.title("Normalized Cut with "+ kernel + " kernel")
plt.xlabel("Accuracy " +  str(np.round((totalAccuracy*100.0/nData),2))+ "%" )
plt.legend(loc='best')
plt.savefig("Report/NormalizedCut_"+kernel+".png", bbox_inches="tight")
#plt.show ()
```

- SUPPORT VECTOR VISUALIZING

Import the libraries and data is the same as another files

```
from svmutil import *
from svm import *
import csv # for CSV reader
import numpy as np
from math import *
import numpy.matlib
from numpy import linalg as LA
import matplotlib.pyplot as plt
import matplotlib.colors
import re

TrainData = []
TrainLabel =[]
nTrain =0
K=5
with open('X_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                tempRow = []
                for i in range (28*28):
                        tempRow.append((float)(row[i]))
                TrainData.append(tempRow)

with open('T_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                TrainLabel.append ((float)(row[0]))
nTrain = len(TrainLabel)
```

Reduce the data to 2 cluster only, in case needed

```
# #reduce data 2 cluster only
# TrainData = np.array(TrainData)
# TrainData = TrainData[0:2000]
# TrainLabel = np.array(TrainLabel)
# TrainLabel = TrainLabel[0:2000]
# nTrain = 2000
# K = 2
```

Calculate the PCA to project data to 2D, the same as anther PCA part

```
mean = np.mean(TrainData,axis=0)
centered=TrainData-np.matlib.repmat(mean,nTrain,1)
covariance=(centered.T.dot(centered))/(nTrain-1)
DP = 2
eigenValues, eigenVectors = np.linalg.eig(covariance)
largestEigenVectorIndex =  np.argsort(eigenValues.real)
largestEigenVectorIndex = largestEigenVectorIndex[::-1][:DP]
PCA =  eigenVectors[:,largestEigenVectorIndex]
W = np.array(TrainData).dot (PCA)
```

Define the marker and color for each fingures

```
color = ['b','g','r','c','m']
marker = ['x','+','1','2','*']
markerX = ['x','+','h','y','*']
```

Set the parameter for svm library. The used parameters is obtained from the homework 5.

```
prob  = svm_problem(list(TrainLabel), list(TrainData ))
linear = 0
polynomial = 1
RBF =2
s = 0
t = 1
g = 0.0625
d = 2
c = 100
```

```
v = 3
```
Run SVM using Polynomial kernel. With the configurations
```
polynomial = True
if (polynomial == True) :
        bestc =  0.5
        bestd = 2
        bestg = 0.25

        param = svm_parameter("-q -t 1 -c " + str(bestc) +" -g " + str (bestg) + " -d "
+ str (bestd) )
        m = svm_train(prob, param)
```
taking the indices of data that is lay on the support vectors
```
        indices = []
        idx = np.array(m.get_sv_indices())
```
Ploting the point that is already projected using PCA
```
        for k in range (K):
                plt.plot (W[k*1000:(k+1)*1000,0],W[k*1000:
(k+1)*1000,1],marker='o',color=color[k],alpha =0.8, markersize=2,linestyle='none',
label="Label " + str(k+1))
```
Plotting Support Vector
```
        for id in idx:
                plt.plot (W[id-1,0],W[id-1,1],marker= marker[0],color='k',alpha
=1,linestyle='none', markersize=4 )
```
Saving figure as image
```
        print (idx.shape[0])
        plt.title("SVM Polynomial PCA Projection")
        plt.legend(loc="best")
        plt.savefig("PolynomialSVM_clusters.png", bbox_inches="tight",dpi=300)
        plt.clf()


TrainData = np.array(TrainData)
```
SVM using Linear kernel. Calculate Linear Kernel manually.
```
# manual T
linearManual =True
if (linearManual == True) :
        KKTrain = TrainData.dot(TrainData.T)
        index = range (1,nTrain+1)
        KKTrain = np.column_stack([index, KKTrain])
        KKTrain = (KKTrain).tolist()
```
Setting parameters for SVM library with the user-defined Kernel
```
        prob  = svm_problem(TrainLabel, KKTrain, isKernel=True)
        param = svm_parameter('-t 4 -c 100 -q')
        m = svm_train(prob, param)
        print("LINEAR MANUAL")
```
Obtain the the indices that are related to the support vectors.
```
        indices = []
        idx = np.array(m.get_sv_indices())
```
Plotting data projtected by PCA
```
        for k in range (K):
                plt.plot (W[k*1000:(k+1)*1000,0],W[k*1000:
(k+1)*1000,1],marker='o',color=color[k],alpha =0.8, markersize=2,linestyle='none',
label="Label " + str(k+1))
```
Plotting Support Vector
```
        for id in idx:
                plt.plot (W[id-1,0],W[id-1,1],marker= marker[1],color='k',alpha
=1,linestyle='none', markersize=4 )
        print (idx.shape[0])
        plt.title("SVM Linear PCA Projection")
        plt.legend(loc="best")
        plt.savefig("LinearSVM_clusters.png", bbox_inches="tight",dpi=300)
        plt.clf()
```

SVM using RBF. Calculate RBF Kernel manually.

```
RBF =True
if (RBF == True) :
```

Calculate Euclide distance

```
        difDist= np.zeros((nTrain,nTrain))
        print ("Calculating diff")
        for i in range (nTrain):
                for j in range (i,nTrain,1):
                        difDist[i,j] = np.linalg.norm( TrainData[i] -TrainData[j])
                        difDist[j,i] = difDist[i,j]
```

calculate Similarity Matrix using RBF kernel

```
        print ("Done Diff")
        KK2Train = np.zeros((nTrain,nTrain))

        for i in range (nTrain):
                for j in range (i,nTrain,1):
                        KK2Train[i,j] =  exp((-g)*difDist[i,j])
                        KK2Train[j,i] =  KK2Train[i,j]

        print ("Done Kernel Train")
        index = range (1,nTrain+1)
        KK2Train = np.column_stack([index, KK2Train])
        KK2Train = (KK2Train).tolist()
```

Setting SVM problem

```
        print("RBF MANUAL")
        prob  = svm_problem(TrainLabel, KK2Train, isKernel=True)
        param = svm_parameter('-t 4 -c 2 -q')
        m = svm_train(prob, param)
```

Obtain the the indices that are related to the support vectors.

```
        indices = []
        idx = np.array(m.get_sv_indices())
```

Plotting data projtected by PCA

```
        for k in range (K):
                plt.plot (W[k*1000:(k+1)*1000,0],W[k*1000:
(k+1)*1000,1],marker='o',color=color[k],alpha =0.8, markersize=2,linestyle='none',
label="Label " + str(k+1))
```

Plotting Support Vector

```
        for id in idx:
                plt.plot (W[id-1,0],W[id-1,1],marker= marker[2],color='k',alpha
=1,linestyle='none', markersize=4 )

        print (idx.shape[0])
        plt.title("SVM RBF PCA Projection")
        plt.legend(loc="best")
        plt.savefig("RBFKernelSVM_clusters.png", bbox_inches="tight",dpi=300)
        plt.clf()
```

Combine Linear + RBF kernel for SVM problem

```
linearRBF = True
if (linearRBF == True) :
#KK1 + KK2
        KK3Train = np.zeros((nTrain,nTrain))
```

Linear combine Linear Kernel and RBF Functions that are already calculated above

```
        for i in range (nTrain):
                for j in range (i,nTrain,1):
                        KK3Train[i,j] = KKTrain[i][j+1] + KK2Train[i][j+1]
                        KK3Train[j,i] =  KK3Train[i,j]

        print ("Done Kernel Train")
        index = range (1,nTrain+1)
        KK3Train = np.column_stack([index, KK3Train])
        KK3Train = (KK3Train).tolist()
```

```
        print ("Done Diff")

        print("LINEAR + RBF MANUAL")
```

Set SVM problem using user-defined Kernel

```
        prob  = svm_problem(TrainLabel, KK3Train, isKernel=True)
        param = svm_parameter('-t 4 -c 0.0156 -q')
        m = svm_train(prob, param)
```

Obtain the the indices that are related to the support vectors.

```
        indices = []
        idx = np.array(m.get_sv_indices())
```

Plotting data projtected by PCA

```
        for k in range (K):
                plt.plot (W[k*1000:(k+1)*1000,0],W[k*1000:
(k+1)*1000,1],marker='o',color=color[k],alpha =0.8, markersize=2,linestyle='none',
label="Label " + str(k+1))
```

Plotting Support Vector

```
        for id in idx:
                plt.plot (W[id-1,0],W[id-1,1],marker= marker[3],color='k',alpha
=1,linestyle='none', markersize=4 )

        print (idx.shape[0])
        plt.title("SVM Linear + RBF PCA Projection")
        plt.legend(loc="best")
        plt.savefig("LInear + RBFKernelSVM_clusters.png", bbox_inches="tight",dpi=300)
        plt.clf()
```

- LDA

Importing necessery libraries

```
import numpy as np
from math import *
import numpy.matlib
from numpy import linalg as LA
import matplotlib.pyplot as plt
import matplotlib.colors
from numpy.linalg import inv
from numpy.linalg import pinv
from sympy import Matrix

TrainData = []
nData =0
```

Loading csv data

```
#TRAIN DATA
with open('X_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                tempRow = []
                for i in range (28*28):
                        tempRow.append((float)(row[i]))
                TrainData.append(tempRow)

TrainData = np.array(TrainData)
nDimesion = TrainData.shape[1]
nData = TrainData.shape[0]
print (TrainData.shape)
print (nData)

TrainLabel = []
with open('T_train.csv') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter=',')
        for row in csv_reader:
                TrainLabel.append ((int)(row[0]))
TrainLabel = np.array(TrainLabel)

nDimesion = 28*28
```

Getting the labels of the cluster and the number of each cluster.

```
(values,counts) = np.unique (TrainLabel, return_counts=True)
K = values.shape[0]
print (K)
```

Saperating the data to K clusters.

```
X_Data = []
for k in range (K):
        index = np.where(TrainLabel==values[k])
        index = np.array(index)
        X_Data.append (TrainData[index].reshape((counts[k],nDimesion)))

X_Data = np.array(X_Data)

print (X_Data.shape)
```

Calculating the mean of each cluster

```
miu = np.zeros((K,nDimesion))
for k in range (K) :
        miu[k] = np.mean (X_Data[k],axis=0)
```

Calculate the total mean, by the mean of the mean.

```
TotalMiu = np.mean (TrainData,axis=0)
```

Calculate Between-class Matrix SBTotal

```
# calculate SB Between Class
SB = np.zeros((K,nDimesion,nDimesion))
SBTotal = np.zeros((nDimesion,nDimesion))
for k in range (K) :
        SB[k] = 1000 * ((miu[k]-TotalMiu).dot((miu[k]-TotalMiu).T))
```

```
        SBTotal += SB[k]
```

Calculate Within-Class Matrix SWTotal. SWTotal is sum of all covariance matrix of each cluster

```
SW = np.zeros((K,nDimesion,nDimesion))
SWTotal = np.zeros((nDimesion,nDimesion))

for k in range (K) :
        centered=X_Data[k]-np.matlib.repmat(miu[k],counts[k],1)
        covariance=(centered.T.dot(centered))
        SWTotal += covariance

#I = exp(-18) * np.identity(nDimesion)
```

Inverse matrix SW, SW might be singular, so we need to do Pseudo inverse

```
SW_Inverse = pinv(SWTotal)
```

Calculate and solve Eigen problem of matrix M = SW^(-1).dot(SB)

$$\arg\max_W \frac{W^T S_B W}{W^T S_W W}$$

$$S_W W = \lambda S_B W$$

```
M=  SW_Inverse.dot(SBTotal)
DP = 2
eigenValues, eigenVectors = np.linalg.eig(M)
```

Choosing the first 2 Eigen value that are related to the first 2 largest eigen values. Form the LDA projection

```
largestEigenVector =  np.argsort(eigenValues.real)
largestEigenVector = largestEigenVector[::-1]

LDA =np.zeros((nDimesion,0))

for i in range(DP) :
        newCoor = (np.array(eigenVectors[:,largestEigenVector[i]]))
        newCoor = np.reshape(newCoor,(nDimesion,1))
        LDA = np.concatenate((LDA, newCoor), axis=1)
```

Project the data to LDA. W = X.dot(LDA)

```
W = np.zeros((K,X_Data.shape[1],DP))
#project to LDA
for k in range (K) :
        w[k] = X_Data[k].dot (LDA.real)

print (V.shape)
```

Visualizing data

```
isVisualize = True
if (isVisualize==True) :
        for k in range (K) :
          plt.plot (V[k,:,0].real,V[k,:,1].real,marker='o',color=plt.cm.tab10(k),alpha =0.8, linestyle='none')
        plt.show()
```

- Eigen Face

Importing necessary libraries

```
import csv # for CSV reader
import numpy as np
from math import *
import numpy.matlib
from numpy import linalg as LA
import matplotlib.pyplot as plt
import matplotlib.colors
import matplotlib.image
import re
import pandas as pd
import time
import numpy.matlib
import matplotlib.gridspec as gridspec
import random
```

Read pgm format

```
def read_pgm(filename, byteorder='>'):
    with open(filename, 'rb') as f:
        buffer = f.read()
    try:
        header, width, height, maxval = re.search(
            b"(^P5\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n]\s)*)", buffer).groups()
    except AttributeError:
        raise ValueError("Not a raw PGM file: '%s'" % filename)
    return np.frombuffer(buffer,
                         dtype='u1' if int(maxval) < 256 else byteorder+'u2',
                         count=int(width)*int(height),
                         offset=len(header)
                         )
```

Read data. Using list of data from files.csv

```
fileName = []
with open('files.csv') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        fileName.append ((row[1]))

fileName= np.array(fileName)
nData = fileName.shape[0]
```

Load all of the face to the list faces. Each face is represented by a vector of size (112*92)

```
faces = []
for i in range (nData):
        img = read_pgm(fileName[i])
        faces.append(img)
faces= np.array(faces)
print (faces.shape)
```

De finding the dimension of data

```
nDimesion = 112*92
```

Calculating the PCA. Firstly, calculate the mean face

```
mean = np.mean(faces,axis=0)
```

Centralized the data by subtract all of the face to the mean face

```
centered=faces-np.matlib.repmat(mean,nData,1)
```

Calculating the covariance matrix.

```
covariance=(centered.T.dot(centered))/(nData-1)
```

Set the new dimension by using PCA is 400 as default. Solving eigen problem of the covariance matrix. And selecting the first K vectors that are related to the first K largest eigen values

```
K =400
start_time = time.time()
print ("calculating eigenValues, eigenVectors")
```

```
eigenValues, eigenVectors = np.linalg.eig(covariance)
print("time for calculating: " + str((time.time() - start_time)/60) +" minutes" )

largestEigenVectorIndex =  np.argsort(eigenValues.real)
largestEigenVectorIndex = largestEigenVectorIndex[::-1]
PCA =np.zeros((nDimesion,0))

for i in range(K) :
        newCoor = (np.array(eigenVectors[:,largestEigenVectorIndex[i]]))
        newCoor = np.reshape(newCoor,(nDimesion,1))
        PCA = np.concatenate((PCA, newCoor), axis=1)
```

The K Principal components is the Eigen faces., Visulize the first 25 eigein faces means that visulzed the first 25 Principal components.

```
isDrawPCA = True
if (isDrawPCA==True) :
    plt.figure(figsize = (5,5))
    gs1 = gridspec.GridSpec(5, 5)
    gs1.update(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
    for i in range (5) :
        for j in range (5) :
            eigenFace =  np.reshape (PCA[:,(i*5+j)],(112,92))
            ax= plt.subplot(gs1[i*5+j])
            ax.axis('off')
            ax.imshow(eigenFace.real,cmap='gray')

    plt.savefig("../Report/eigenFace.png", bbox_inches="tight", dpi=150)
    plt.clf()
```

Reconstructing the face using Principal Components. Using different number of Principal component to give a comparison.

Cancelate the reconstruct face by the formula. Firstly, projecting the original data to new space.

$$W = CenteredData.dot(PCA)$$

Then reconstruct the data

$$ReconstructedData = W.dot(PCA.T) + mean$$

```
#reconstruct face
W=centered.dot(PCA)
Reconstruct  = W.dot(PCA.T) + mean
print (PCA.shape)
```

Visualizing data by toggle the variable `isDrawReconstructedFace`

```
isDrawReconstructedFace = True
if (isDrawReconstructedFace) :

    nFace = 25
    idx = random.sample(range(1, nData-1), nFace)
    #fig3, ax3 = plt.subplots(nrows=5, ncols=5)
    plt.figure(figsize = (5,5))
    gs1 = gridspec.GridSpec(5, 5)
    gs1.update(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
    for i in range (5) :
        for j in range (5) :
            ReconstructedImg=  np.reshape (Reconstruct[idx[i*5+j]],(112,92))
            ax= plt.subplot(gs1[i*5+j])
            ax.axis('off')
            ax.imshow(ReconstructedImg.real,cmap='gray')

    plt.savefig("../Report/Reconstructed_400_Face.png", bbox_inches="tight", dpi=150)
    plt.clf()
```

Visualizing the original Faces

```
    for i in range (5) :
        for j in range (5) :
            faceImg=  np.reshape (faces[idx[i*5+j]],(112,92))
```

```
            ax= plt.subplot(gs1[i*5+j])
            ax.axis('off')
            ax.imshow(faceImg.real,cmap='gray')

    plt.savefig("../Report/Original_400_Face.png", bbox_inches="tight", dpi=150)
    plt.clf()
```

Reduce the number of used Principal Component to **200** and visualizing

```
    DP = 200
    PCA = PCA[:,:DP]
    W=centered.dot(PCA)
    Reconstruct  = W.dot(PCA.T) + mean

    for i in range (5) :
        for j in range (5) :
            ReconstructedImg=  np.reshape (Reconstruct[idx[i*5+j]],(112,92))
            ax= plt.subplot(gs1[i*5+j])
            ax.axis('off')
            ax.imshow(ReconstructedImg.real,cmap='gray')

    plt.savefig("../Report/Reconstructed_"+str(DP)+"_Face.png", bbox_inches="tight",
dpi=150)
    plt.clf()
```

Reduce the number of used Principal Component to **100** and visualizing

```
    DP = 100
    PCA = PCA[:,:DP]
    W=centered.dot(PCA)
    Reconstruct  = W.dot(PCA.T) + mean

    for i in range (5) :
        for j in range (5) :
            ReconstructedImg=  np.reshape (Reconstruct[idx[i*5+j]],(112,92))
            ax= plt.subplot(gs1[i*5+j])
            ax.axis('off')
            ax.imshow(ReconstructedImg.real,cmap='gray')

    plt.savefig("../Report/Reconstructed_"+str(DP)+"_Face.png", bbox_inches="tight",
dpi=150)
    plt.clf()
```

Reduce the number of used Principal Component to **25** and visualizing

```
    DP = 25
    PCA = PCA[:,:DP]
    W=centered.dot(PCA)
    Reconstruct  = W.dot(PCA.T) + mean

    for i in range (5) :
        for j in range (5) :
            ReconstructedImg=  np.reshape (Reconstruct[idx[i*5+j]],(112,92))
            ax= plt.subplot(gs1[i*5+j])
            ax.axis('off')
            ax.imshow(ReconstructedImg.real,cmap='gray')

    plt.savefig("../Report/Reconstructed_"+str(DP)+"_Face.png", bbox_inches="tight",
dpi=150)
    plt.clf()
```