Machine Learning – Homework 7 (Final Homework)

I – Kernel eigenfaces / fisherfaces

Code

I have three different python files, one concerning LDA/Kernel LDA, another one concerning PCA and kernel PCA and finally a last one that uses the other one in order to do the testing and see the performance.

<u>First file : PCA</u>

```python
from skimage import io
import os
import numpy as np
from skimage.transform import resize
from scipy.spatial import distance


path = {"Training":"Yale_Face_Database/Training/",
        "Test":"Yale_Face_Database/Testing/"}
```

First we import the libraries we will use in our program. Skimage is helpful to load the images and resize them into to increase the computation speed. I then defined a path variable to locate the training and testing images.

```python
def create_data_set(data_path):
        """
        Load the data based on a data path.
        It returns a 3 dimension vector X containing the images, and a label vector containing the filenames
        Be careful : They are not sorted as in the directory.
        """
        images_path = [ os.path.join(data_path, item)  for item in  os.listdir(data_path) ]
        image_data = []
        image_labels = [item for item in os.listdir(data_path)]

        for i,im_path in enumerate(images_path):
                im = io.imread(im_path,as_gray=True)
                im = resize(im,(im.shape[0]//2,im.shape[1]//2), anti_aliasing=True)
                image_data.append(im)


        X = np.array(image_data).astype('float32')
        return X, image_labels
```

This function basically load the images into a variable X. I will use it twice to load both the training and testing data. It also returns the filenames associated, so that I can construct a label vector which will contain the classes.

```python
def compute_mean(X,Y):
        """
        Takes parameter X containing the images as a tensor, and a vector Y which contains the label of each training example.
        Then reshape the images into a vector, to compute two means : One is the mean of each class, and global mean which is the
        mean of all the training examples.
        """
        mean = np.zeros((15,X.shape[1]*X.shape[2]))
        for i in range(X.shape[0]):
                mean[Y[i]] += X[i].reshape((-1))/9
        global_mean = np.mean(mean,axis=0)
        return mean, global_mean

```

This function aims to compute the mean of an X data set, with the help of the label vector. It will return two different means : a mean associated to every class, so this one have the size of the number of classes and uses the trainings of each classes to compute its own mean. Then another mean is the global mean, which doesn't care about the label and takes all the examples to compute a global mean.

```python
def get_feature_vectors(covariance):
        """
        Return the feature vectors associated with the 25th highest eigenvalues
        """
        eigen_values, eigen_vectors = np.linalg.eigh(covariance)
        idx = eigen_values.argsort()[::-1]
        return eigen_vectors[:,idx][:,:25]

```

The get feature vectors function takes only the covariance matrix for the PCA. Or a kernel matrix. It computes the eigenvectors and eigenvalues thanks to numpy. Then it return the eigenvectors associated to the 25 highest eigenvalues. So a base for our low dimension space.

```python
def rbf_kernel(x,y,gamma):
        temp = distance.cdist(x,y,'euclidean')
        return np.exp(-gamma*temp)
def linear_kernel(x,y,c):
        res = x@y.T + c
        return res
def rq_kernel(x1,x2,param=[1,1,1]):
        """
                rational quadratic kernel, 3 parameters : sigma,alpha,l
        """
        l,sigma,alpha = param
        temp = distance.cdist(x1,x2,'euclidean')
        return sigma**2*(1+temp/(2*alpha*l**2))**(-alpha)
```

These three functions are taken from older homeworks, it just compute the kernel matrix, in different ways. I have 3 kernels here, the linear kernel, the rbf kernel and the rational quadratic kernel.

```python
def PCA(X_train,Y_train,kernel='None'):
        """
        This function is made to be used when the file is imported in another function.
        Return the eigenvectors from PCA, right now 25 eigenvectors.
        CAn be modified by changing the above function.
        """
        print("Computing mean and global mean ...")
        mean, global_mean = compute_mean(X_train,Y_train)
        print("Done.")
        print("Computing covariance matrix ...")
        x = X_train.reshape((X_train.shape[0],-1))-global_mean
        if kernel=='linear':
                covariance = linear_kernel(x.T,x.T,1)
        if kernel=='RQ':
                covariance = rq_kernel(x.T,x.T)
        if kernel=='RBF':
                covariance =  rbf_kernel(x.T,x.T,0.1)
        if kernel =='None':
                covariance = np.cov((X_train.reshape((X_train.shape[0],-1))-global_mean).transpose())
        print("Done.")
        print("Computing feature vectors ...")
        feature_vectors = get_feature_vectors(covariance) # 11155 by 25
        print("Done.")
        return feature_vectors

```

This is a function that uses a lot of other function I have already defined. It is made to be used by another python program. Basically, it takes the training set and label, and an optionnal kernel, and it returns the eigenvectors by applying PCA.

```python
if __name__=="__main__":
        ####################  PCA ################################
        X_train, label_train = create_data_set(path["Training"])
        Y_train = [int(x[7:9])-1 for x in label_train]
        X_test, label_test = create_data_set(path["Test"])
        mean, global_mean = compute_mean(X_train,Y_train)
        covariance = np.cov((X_train.reshape((X_train.shape[0],-1))-global_mean).transpose()) # 11155 by 11155
        feature_vectors = get_feature_vectors(covariance) # 11155 by 25
        random_indexes = np.random.randint(low=0,high=X_train.shape[0],size=10)
        reconstructed_images = []
        for i in random_indexes:
                projected_image = np.matmul(X_train[i].reshape((-1)),feature_vectors)
                temp = global_mean.reshape((115,97))+np.matmul(projected_image,feature_vectors.transpose()).reshape((115,97))
                io.imsave("Results/rec_"+label_train[i],temp)
                io.imsave("Results/original_"+label_train[i],X_train[i])
                reconstructed_images.append(temp)
```

Main function, that I used to apply PCA to the training set and then reconstruct the images, before saving everything in a result folder. It follow the PCA steps and use the above function, but not the PCA function because I hadn't write it before I created my third python program. This snippet can't be used by another program, contrary to the PCA function. But it does similar things.

<u>Second file : LDA and Kernel Lda</u>

```python
from skimage import io
import os
import numpy as np
from skimage.transform import resize
from scipy.spatial import distance

path = {"Training":"Yale_Face_Database/Training/",
        "Test":"Yale_Face_Database/Testing/"}

```

Exactly the same thing as for the first file for PCA.

```python
def create_data_set(data_path):
	"""
	Load the data based on a data path.
	It returns a 3 dimension vector X containing the images, and a label vector containing the filenames
	Be careful : They are not sorted as in the directory.
	"""
	images_path = [ os.path.join(data_path, item)  for item in  os.listdir(data_path) ]
	image_data = []
	image_labels = [item for item in os.listdir(data_path)]

	for i,im_path in enumerate(images_path):
		im = io.imread(im_path,as_gray=True)
		im = resize(im,(im.shape[0]//3,im.shape[1]//3), anti_aliasing=True)
		image_data.append(im)

	X = np.array(image_data).astype('float32')
	return X, image_labels
```

```python
def compute_mean(X,Y):
	"""
	Takes parameter X containing the images as a tensor, and a vector Y which contains the label of each training example.
	Then reshape the images into a vector, to compute two means : One is the mean of each class, and global mean which is the
	mean of all the training examples.
	"""
	mean = np.zeros((15,X.shape[1]*X.shape[2]))
	for i in range(X.shape[0]):
		mean[Y[i]] += X[i].reshape((-1))/9
	global_mean = np.mean(mean,axis=0)
	return mean, global_mean
```

Again, it has the same purpose and these are exactly the same functions. I could've used them since they are already defined in the PCA file. But for interpreted command line it was more usable to just copy paste it in my LDA file.

```python
def within_class_matrix(x,y, mean):
	"""
	Compute the within class scatter matrix.
	"""
	w_c_mat = np.zeros([x.shape[1]*x.shape[2], x.shape[1]*x.shape[2]], dtype=np.float32)
	for i in range(0, x.shape[0]):
		temp = np.subtract(x[i].reshape((-1)), mean[y[i]])
		temp = temp.reshape((-1,1))
		w_c_mat += np.matmul(temp, temp.transpose())
	return w_c_mat
```

This one is a new function for the LDA algorithm. It compute the within class scatter matrix, which is a representation of the distribution of the data inside each class, but ignore the relation of a data point with other class points.

```python
def between_class_matrix(mean, global_mean):
	"""
	Compute the between class scatter matrix
	"""
	b_c_mat = np.zeros([mean.shape[1], mean.shape[1]], dtype=np.float32)
	for i in range(0, 9):
		temp = np.subtract(mean[i], global_mean).reshape(mean.shape[1], 1)
		b_c_mat += np.matmul(temp, temp.transpose())
		b_c_mat *= 9
	return b_c_mat
```

And this is the between class matrix function, which represents how the different class are related. We want to maximize the difference between every class in the lower dimension space, while minimizing the variance inside a class. This is the problem we are trying to solve in LDA.

```python
def LDA(X_train,Y_train):
        """
        Return the feature eigenvectors, right now it is 25 eigenvectors. Can be changed by
        modifying the get_feature_vector function.
        """
        print('Computing means...')
        mean, global_mean = compute_mean(X_train,Y_train)
        print('Done.')
        temp = X_train.reshape((X_train.shape[0],-1))-global_mean
        X_train = temp.reshape((X_train.shape))
        print('Computing within class matrix ...')
        w_c_mat = within_class_matrix(x=X_train,y=Y_train,mean=mean)
        print('Done.')
        print('Computing between class matrix ...')
        b_c_mat = between_class_matrix(mean=mean, global_mean=global_mean)
        print('Done.')
        print('Computing feature vectors ...')
        feature_vectors = get_feature_vectors(w_c_mat,b_c_mat)
        print('Done.')
        return feature_vectors
```

This is the function that will be used in another python program, in order to run LDA on a training set. It returns the main eigenvectors, which represent the lower dimension space. It uses the functions defined above in order to compute the eigenvectors of the space that fit the best the problem of LDA that I just talked about above.

```python
def rbf_kernel(x,y,gamma):
        temp = distance.cdist(x,y)
        return np.exp(-gamma*temp)

def linear_kernel(x,y,c):
        res = x@y.T + c
        return res

def rq_kernel(x1,x2,param=[1,1,1]):
        """
        rational quadratic kernel, 3 parameters : sigma,alpha,l
        """
        l,sigma,alpha = param
        temp = distance.cdist(x1,x2)
        return sigma**2*(1+temp/(2*alpha*l**2))**(-alpha)
```

Define the kernel functions for the kernel LDA.

```python
def K_LDA(X_train,Y_train,kernel='RBF'):
    """
        Kernel LDA, instead of the basic within class matrix and the between class matrix,
        we compute two corresponding matrix based on the kernel we first compute.
    """
    print('Computing means...')
    mean, global_mean = compute_mean(X_train,Y_train)
    print('Done.')
    x = X_train.reshape((X_train.shape[0],-1))-global_mean
    if kernel=='RBF':
        K = rbf_kernel(x.T,x.T,gamma=0.1)
    if kernel=='linear':
        K = linear_kernel(x.T,x.T,1)
    if kernel == 'RQ':
        K = rq_kernel(x.T,x.T)

    index = {i:[] for i in range(15)}
    for i in range(len(Y_train)):
        index[Y_train[i]].append(i)
    Ks = {i:[] for i in range(15)}
    for i in K:
        for key,val in index.items():
            temp = []
            for h in val:
                temp.append(i[h])
            Ks[key].append(np.array(temp))
    for key in Ks.keys():
        Ks[key] = np.asarray(Ks[key])

    A = np.identity(9) - ((1/float(9)) * np.ones((9,9)))

    print('Compute within class matrix ...')
    # calculate within class scatter matrix N
    N = np.zeros(K.shape)
    for value in Ks.values():
        temp = np.dot(A,value.T)
        temp = np.dot(value, temp)
        N += temp
    print('Done.')

    print('Compute between class matrix ...')
    # calculate M1 and M2
    M = {i:[] for i in range(15)}
    for key,value in Ks.items():
        for i in range(len(value)):
            M[key].append(np.sum(value[i])/float(9))
    for key in M:
        M[key] = np.asarray(M[key])

    Mstar = []
    for i in range(5005):
        Mstar.append(np.sum(value[i])/float(9*15))
    Mstar = np.asarray(Mstar)

    finalM = np.zeros((5005,5005))
    for i in range(15):
        finalM += 9*np.outer((M[i]-Mstar),(M[i]-Mstar).T)
    print('Done.')

    w_c_mat = N
    b_c_mat = finalM
    print('Compute feature vectors ...')
    feature_vectors = get_feature_vectors(w_c_mat,b_c_mat) # 11155 by 25
    print('Done.')
    return feature_vectors
```

This is the kernel LDA functions. Instead of computing the within and between class matrix directly with the data, it uses the kernel matrix to compute two matrix that plays these respective roles. Then it uses the same process of computing eigenvectors to have our lower dimensionnal space. This functions returns again the eigenvectors that we want to project our data on.

```python
if __name__=="__main__":
    ###################  LDA ##############################
    X_train, label_train = create_data_set(path["Training"])
    Y_train = [int(x[7:9])-1 for x in label_train]
    X_test, label_test = create_data_set(path["Test"])
    mean, g X_train: ndarray te_mean(X_train,Y_train)
    temp = X_train.reshape((X_train.shape[0],-1))-global_mean
    X_train = temp.reshape((X_train.shape))
    w_c_mat = within_class_matrix(x=X_train,mean=mean)
    b_c_mat = between_class_matrix(mean=mean, global_mean=global_mean)
    feature_vectors = get_feature_vectors(w_c_mat,b_c_mat) # 11155 by 25
    random_indexes = np.random.randint(low=0,high=X_train.shape[0],size=10)
    reconstructed_images = []
    for i in random_indexes:
            projected_image = np.matmul(X_train[i].reshape((-1)),feature_vectors)
            temp = global_mean.reshape((X_train.shape[1],X_train.shape[2]))+np.matmul(projected_image,feature_vectors.transpose())
            io.imsave("Results_LDA/rec_"+label_train[i],temp)
            io.imsave("Results_LDA/original_"+label_train[i],X_train[i]+global_mean.reshape((X_train.shape[1],X_train.shape[2])))
            reconstructed_images.append(temp)

```

Finally, this is the snippet I used to compute the first question. It finds the 25 eigenvectors, project the data onto it and rebuilt the images before saving them into a directory of results that you can find in my zip file.

Third file : face recognition

I still have quite the same libraries, and basic function to load data, compute mean. So I won't post them again here. I will post what's new.

```python
def knn(X_test, X_train, Y_train,k=20):
    """
    K neareste neighbors algorithm :
        For every points, we first compute its distance to every other points.
        Then we take K closest neighbors, we check their class, and predict the most appeared class for the new point.
    """
    predictions = []
    for current_test in X_test:        # Loop over all the examples
        distances = []
        for current_train in X_train:
            distances.append(distance.euclidean(current_test.reshape((-1)),current_train.reshape((-1))))
        min_liste = n_small_element(distances,k)
        closest_classes = [Y_train[distances.index(i)] for i in min_liste]
        predictions.append(max(set(closest_classes),key=closest_classes.count))
    return predictions


def n_small_element(L,n):
    """
    Helper function to get the K smallest elements of the distance list created in KNN.
    """
    ele = []
    myList = list(np.copy(L)) # To avoid modifying our list with which we call the function.
    for i in range(n):
        ele.append(min(myList))
        myList.remove(min(myList))
    return ele

```

We have here the knn function, that made the predicition of the face recognition. The second function is a helper function in order to get the smallest distance of the distance list. Knn basically compute for every testing points its distance to the trainig points. Then it consider only the K lowest distance, and count how many belongs to every class. The class which is the most represented in these K points, is predicted for the test point.

```python
# FIRST, load the data and create the class vector Y_train. Same for Y_test in order to get the accuracy.
X_train, label_train = create_data_set(path["Training"])
Y_train = [int(x[7:9])-1 for x in label_train]
X_test, label_test = create_data_set(path["Test"])
Y_test = [int(x[7:9])-1 for x in label_test]
# Then compute the mean and global mean which are used in both PCA and LDA.
mean, global_mean = compute_mean(X_train,Y_train)

"""
    X_train         a 3D_tensor containing the training images
    label_train     the filenames of the training images
    Y_train         a class for the images, starting from 0 to 14
    Same for the test variable.
"""
method = 'LDA'
# Comment or uncomment whether you want to use basic LDA or kernel LDA.
if method=='LDA':
    import LDA_eigenfaces
    eigenvectors = LDA_eigenfaces.LDA(X_train,Y_train)
    #eigenvectors = LDA_eigenfaces.K_LDA(X_train,Y_train,kernel='RBF')

if method=='PCA':
    import PCA_eigenfaces
    eigenvectors = PCA_eigenfaces.PCA(X_train,Y_train,'linear')
```

Let me explain this part of the code :

At first, I create the data variable, and labels. Then I compute the mean and global_mean that are useful for PCA and LDA.

Then according to the value of method, I use PCA or LDA to compute the eigenvectors. I can adjust in there whether I want to use Kernel PCA, or Kernel LDA, or simple PCA and LDA. I can also choose which kernel I want among RBF, Rational Quadratic and Linear one.

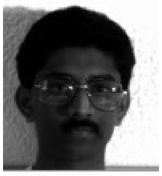
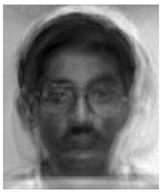
```python
# We now have the eigenvectors to reduce the dimension.
# PS : 25 eigenvectors.
# Now let's project our data into low_dimension space
X_train_reduced = np.matmul(X_train.reshape((X_train.shape[0],-1))-global_mean,eigenvectors)
# Our X_reduced is of dimension (number of training ex) by (number of eigenvectors)
X_test_reduced = np.matmul(X_test.reshape((X_test.shape[0],-1))-global_mean,eigenvectors)

# Then check our results on the testing set.
res = []
ks = [i for i in range(1,26)]
for k in ks:
    pred = knn(X_test_reduced,X_train_reduced,Y_train,k=k)
    correct = 0
    for i in range(len(Y_test)):
        if Y_test[i]==pred[i]:
            correct+=1
    print("Correct : ",correct)
    res.append(correct)
res = np.asarray(res)/30*100
plt.plot(ks,res,'or')
plt.title('Accuracy for different K-nn with LDA')
plt.xlabel('K')
plt.ylabel('Accurcay')
```

Once I have the eigenvectors, this part of the code project the data in the reduced dimension space. Then it applies knn and compute the accuracy of my results. It applies knn with different values of k to see the results.

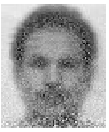
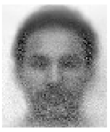
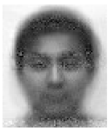
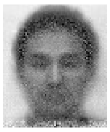
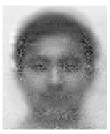
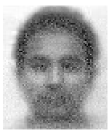
RESULTS

Results of PCA for the reconstructed images :

				
				
				*original images (9 images)*
				*reconstructed images (first projected then reconstructed)*

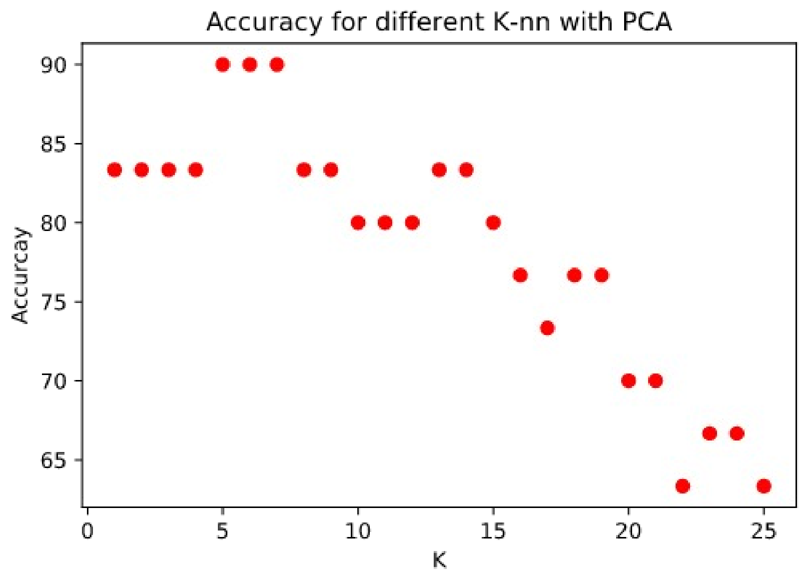
We have some quite nice results here. We see that according to the original face, we have some difference in the reconstruction of the face. That shows that the main informations are well contained into the eigenvectors of the lower dimension space.

<u>Resulsts of LDA for the reconstructed images :</u>

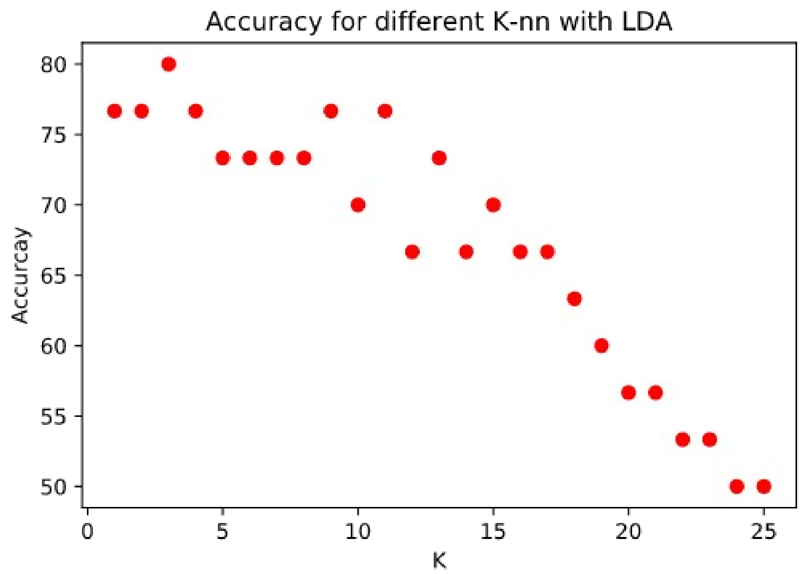
I was less impressed by the results of LDA for the reconstructed faces. We see less difference between the faces. It looks more like an uniform face. Also, There seem to be a noise on the images, like some tiny white points.

Now, the results of the facial recognition :



This is the accuracy of my prediciton for different values of K. We see that our algorithm performs best for K between 5 and 8. We have an accuracy over 90 % which is really nice !! The faces are well recognized in a different mood than the mood presented in the training set.

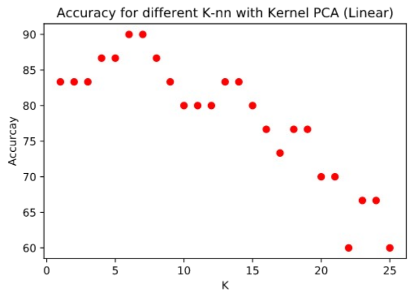
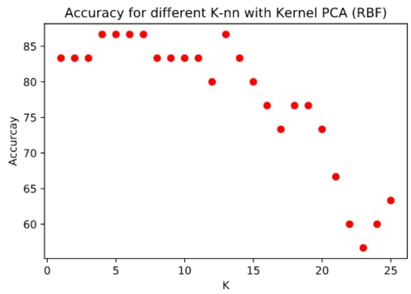
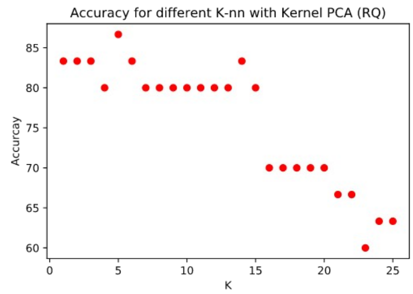
For LDA, we have the following results :



These are the results for LDA. We see that as for the images, the results looks less good. Still we have over 80 % accuracy for K=3 which is good. But overall, the results are less good than for PCA.

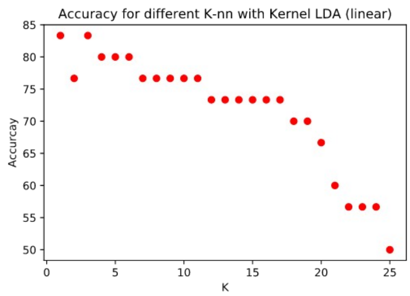
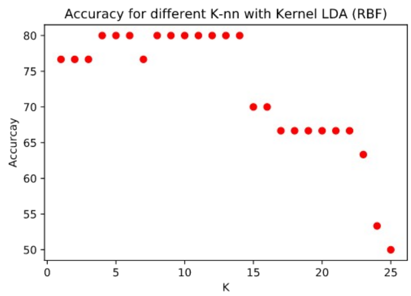
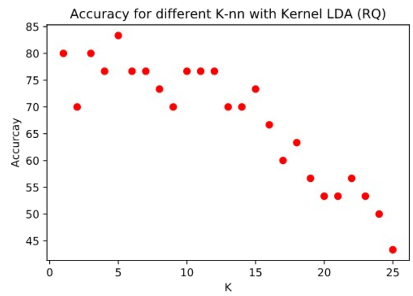
<u>Using different kernels :</u>

Kernel PCA

Linear kernel	Radial basis function kernel	Rational quadratic kernel
		

I didn't figure any significant improvement with kernel PCA. The max accuracy is still around 90 %. And linear kernel performs best with the parameter I chose. We still see that with kernel PCA it works very well.

Kernel LDA

Linear kernel	RBF kernel	Rational quadratic kernel
		

Like for PCA, not any significant improvement using kernel. I even find that the computation were slower for Kernel LDA compared to simple LDA. For RBF kernel we see that the accuracy is still very good even K growing, but overall less efficient. We reach up to 85 % accuracy with both linear and rational quadratic kernels.

LDA might be more efficient in terms of computation speed than PCA. Which makes it better for a real time facial recognition. Kernel PCA and LDA can find a non-linear subspace, which might be better depending on the data we're working on. Basically it gets the data into a higher dimensional space before getting it back into a lower dimensional space in order to find some non-linear curve to project the data on. It is hard to visualize in our case because the dimension number is too high to be plotted.

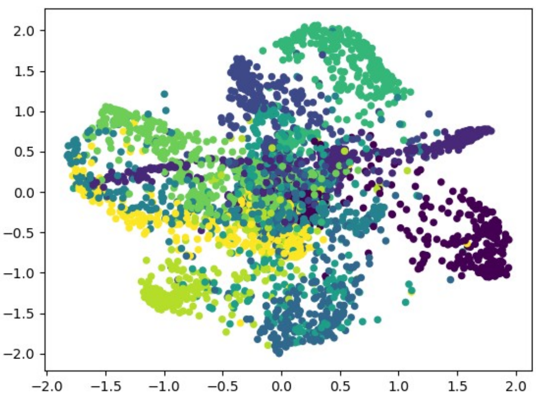
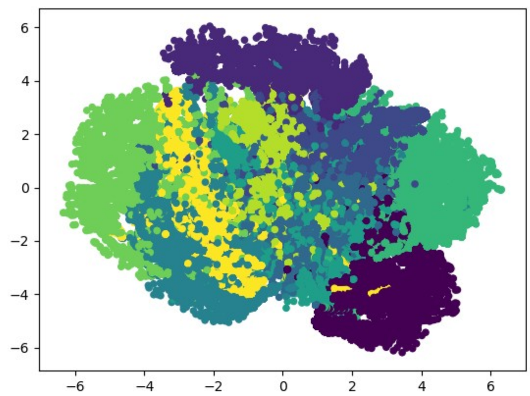
II – t-SNE and s-SNE

Code

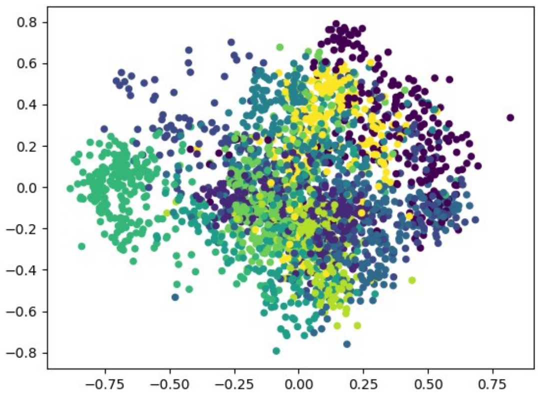
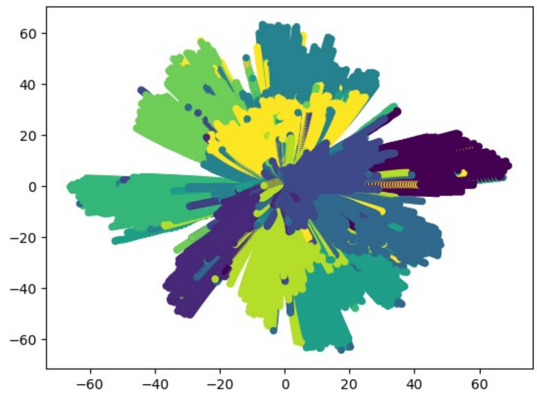
```
142     # Run iterations
143     for iter in range(max_iter):
144 
145         # Compute pairwise affinities
146         sum_Y = np.sum(np.square(Y), 1)
147         num = -2. * np.dot(Y, Y.T)
148         #num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))  #t-sne
149         num = np.exp(-1.*np.add(np.add(num,sum_Y).T,sum_Y))          #s-sne
150         num[range(n), range(n)] = 0.
151         Q = num / np.sum(num)
152         Q = np.maximum(Q, 1e-12)
153 
154         # Compute gradient
155         PQ = P - Q
156         for i in range(n):
157             #dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
158             dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

We can see the difference between t-SNE and s-SNE by checking the line 148 and 149. The operation switches to an exponentiel for s-SNE, which change the conditional probability distribution. And I also changed slightly the gradient on line 157-158, based on the course formula.

Results

Iteration 20	Iteration 500
	

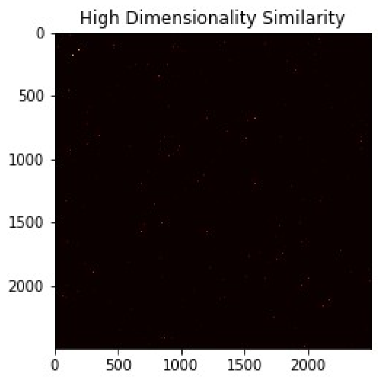
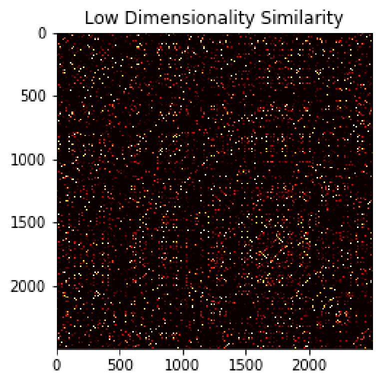
We attend here the dimension reduction in 2D, we clearly see after 500 iteration that there are separate clusters. But we also feel like the clusters are mixing each other, which refers to the crowding problem. So we would like to use t-SNE in order to reduce this crowding problem.

Iteration 10	Iteration 450
	

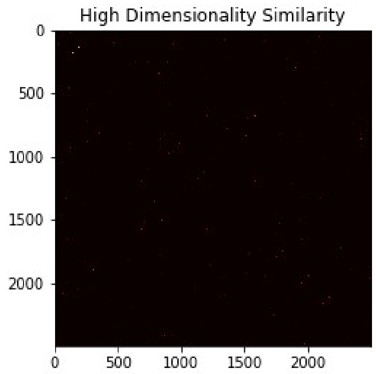
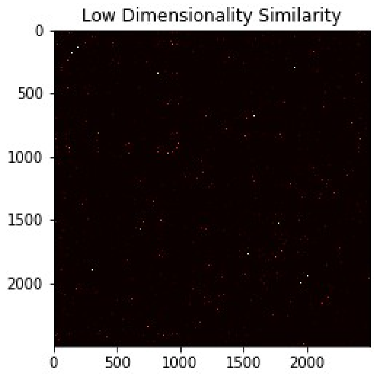
With t-SNE, we see that the crowding problem seems fixed. We have clusters that clearly appears, in separate direction.

You can find in the zip file the gifs that I made from the images to describe the optimization process.

<u>Similarities :</u>

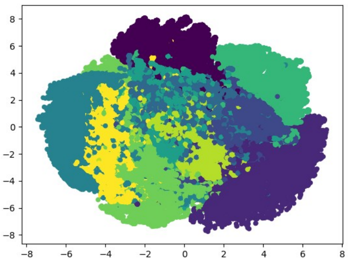
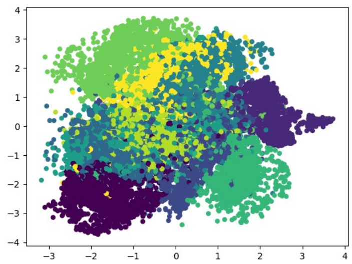
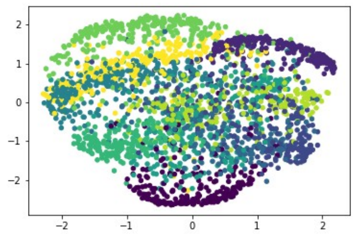
	

This visualization expose again the crowding problem. We see that in high dimension, since we are in the original dimension space, data are not too similar. But getting in the low dimensional space, with s-SNE, we have an overcrowding described by the fact that the points are getting really close from each other, as people are close from each other in a crowd. See below how it improves with t-SNE :

This time, we see that even in low dimension, the points are really less similar than with s-SNE. So we have clusters that are separated from each other as we want to. We managed to keep the similarity in a way that it is close from high dimension similarity. Goal reached !

Perplexity :

20	100	200
		

As the perplexity grows, the shape seems to be more clear and more expanded. It can be usefull to better visualize the data.