Machine Learning Homework 7

Kernel Eigenfaces, t-SNE

ChiaLiang Kuo 310552027
Github :

I. Kernel Eigenfaces

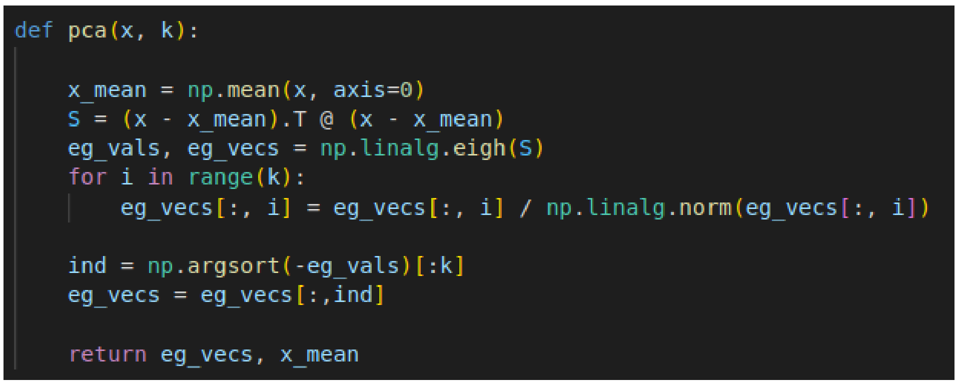
Code and Experiments Results

1. **PCA , LDA**

For PCA, we are going to find the projection matrix W and find 25 first eigenvectors with largest eigenvalues, where the eigenvector is also named as eigenfaces. The x in the function below is a N*(W*H) matrix, where N stands for the number of images and W and H are the width and height of the image. Here I provide 2 versions of PCA implementation which will get the same results. The key idea is that A@A_T and A_T@A have the same non-zero eigenvalues.

First implementation of PCA:

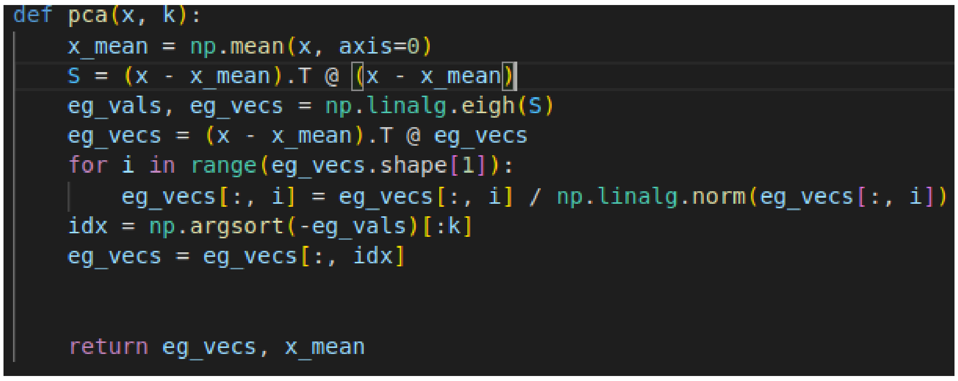
$$A^TAw = \lambda w, \text{w is what we want to get}$$



```python
def pca(x, k):

    x_mean = np.mean(x, axis=0)
    S = (x - x_mean).T @ (x - x_mean)
    eg_vals, eg_vecs = np.linalg.eigh(S)
    for i in range(k):
        eg_vecs[:, i] = eg_vecs[:, i] / np.linalg.norm(eg_vecs[:, i])

    ind = np.argsort(-eg_vals)[:k]
    eg_vecs = eg_vecs[:,ind]

    return eg_vecs, x_mean
```

Second implementation of LDA:

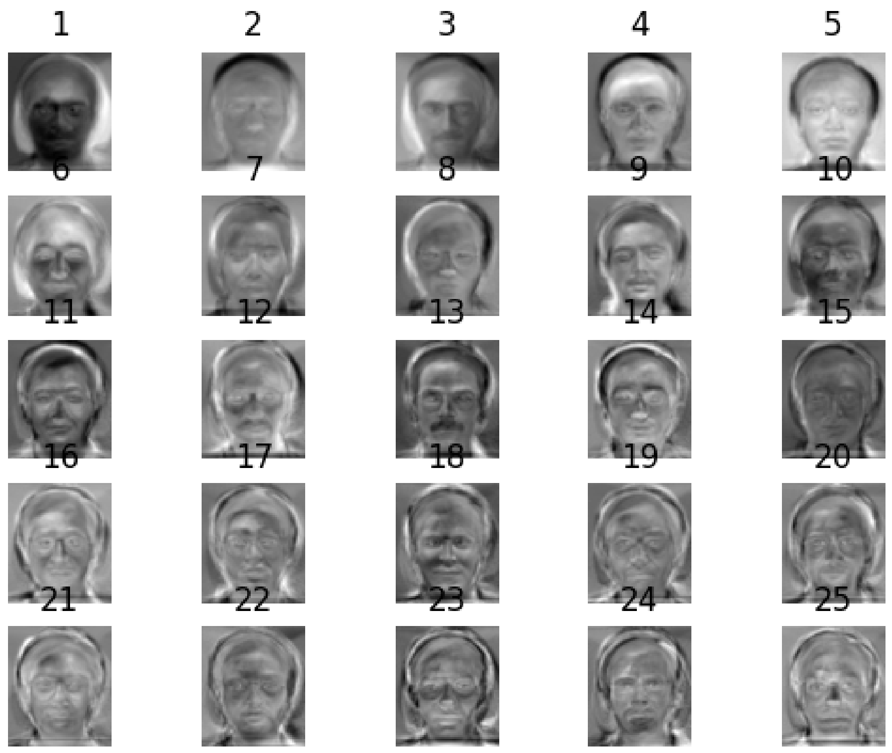
$$AA^Tv = \lambda v$$
$$A^TAA^Tv = \lambda A^Tv$$
$$w = A^Tv, \text{w is what we want to get}$$



```python
def pca(x, k):
    x_mean = np.mean(x, axis=0)
    S = (x - x_mean).T @ (x - x_mean)
    eg_vals, eg_vecs = np.linalg.eigh(S)
    eg_vecs = (x - x_mean).T @ eg_vecs
    for i in range(eg_vecs.shape[1]):
        eg_vecs[:, i] = eg_vecs[:, i] / np.linalg.norm(eg_vecs[:, i])
    idx = np.argsort(-eg_vals)[:k]
    eg_vecs = eg_vecs[:, idx]


    return eg_vecs, x_mean
```

The first implementation compute the eigenvectors by the covariance matrix (X - mean).T @ (X - mean) of size (W*H, W*H). Due to the time complexity issue and the capacity issue, we can use more efficient approach : first compute the eigenvectors of matrix (X - mean) @ (X - mean).T of size N*N, then get the true eigenfaces by computing (x - mean).T @ eigenvectors.

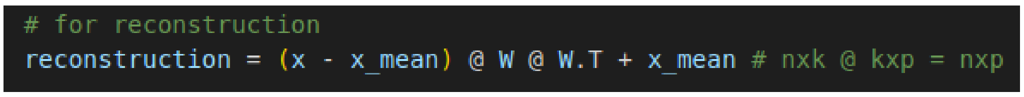
The 25 eigenfaces are shown below :



Face reconstruction for random 10 images :

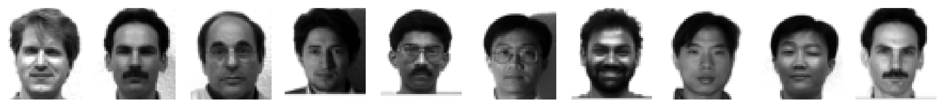
We can reconstruct the face by the code snippet below. Which will first reduce the dimension from W*H to 25 then recover to W*H by using the same eigenvectors set.

$\underbrace{xW}_{z}\underline{W^\top}$

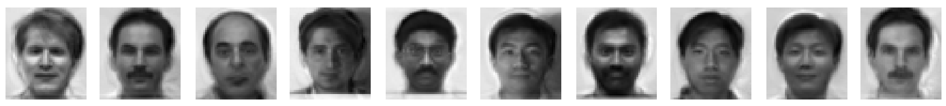


```
# for reconstruction
reconstruction = (x - x_mean) @ W @ W.T + x_mean # nxk @ kxp = nxp
```

true faces



reconstructed faces



In LDA, we will also use eigenvectors corresponding to the first 25 largest eigenvalues to compute the fisherfaces and do the face reconstruction process.

In order to compute the W matrix like what PCA does, we need to compute two scatters : within-class scatter and between-class scatter:

within-class scatter:

$$S_W = \sum_{j=1}^{k} S_j, \text{ where } S_j = \sum_{i \in C_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$$

$$\text{and } \mathbf{m}_j = \frac{1}{n_j} \sum_{i \in C_j} x_i$$

between-class scatter:

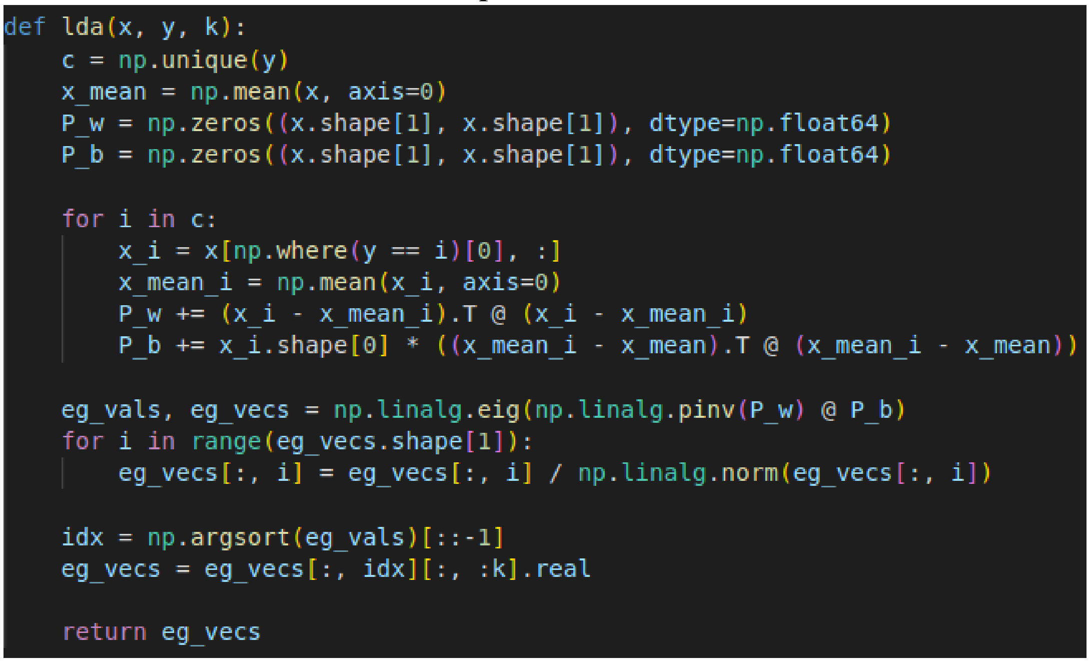
$$S_B = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

$$\text{where } \mathbf{m} = \frac{1}{n} \sum x$$

and then we will get :

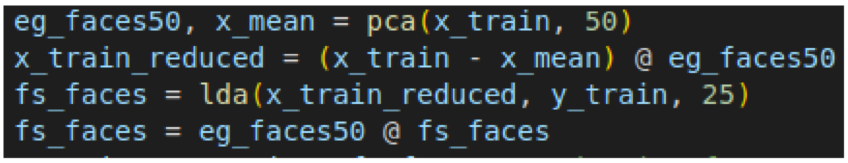
$$W = \text{eigenvectors of } (S_W^{-1} S_B)$$

The full implementation of LDA:



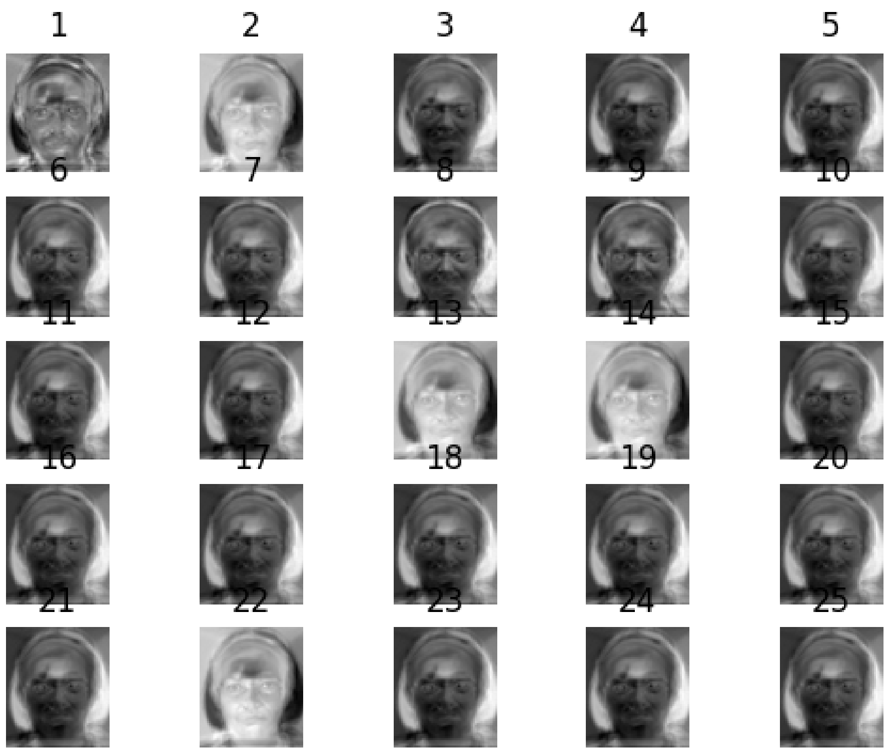
```python
def lda(x, y, k):
    c = np.unique(y)
    x_mean = np.mean(x, axis=0)
    P_w = np.zeros((x.shape[1], x.shape[1]), dtype=np.float64)
    P_b = np.zeros((x.shape[1], x.shape[1]), dtype=np.float64)

    for i in c:
        x_i = x[np.where(y == i)[0], :]
        x_mean_i = np.mean(x_i, axis=0)
        P_w += (x_i - x_mean_i).T @ (x_i - x_mean_i)
        P_b += x_i.shape[0] * ((x_mean_i - x_mean).T @ (x_mean_i - x_mean))

    eg_vals, eg_vecs = np.linalg.eig(np.linalg.pinv(P_w) @ P_b)
    for i in range(eg_vecs.shape[1]):
        eg_vecs[:, i] = eg_vecs[:, i] / np.linalg.norm(eg_vecs[:, i])

    idx = np.argsort(eg_vals)[::-1]
    eg_vecs = eg_vecs[:, idx][:, :k].real

    return eg_vecs
```

In practice, I will first reduce the image dimension to 50 by PCA, and then use both the eigenvectors of PCA and LDA to compute the true fisherfaces. See the code snippet below:

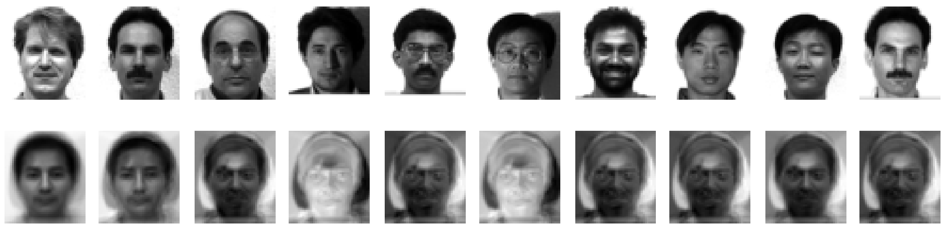


```python
eg_faces50, x_mean = pca(x_train, 50)
x_train_reduced = (x_train - x_mean) @ eg_faces50
fs_faces = lda(x_train_reduced, y_train, 25)
fs_faces = eg_faces50 @ fs_faces
```

The 25 eigenfaces are shown below :

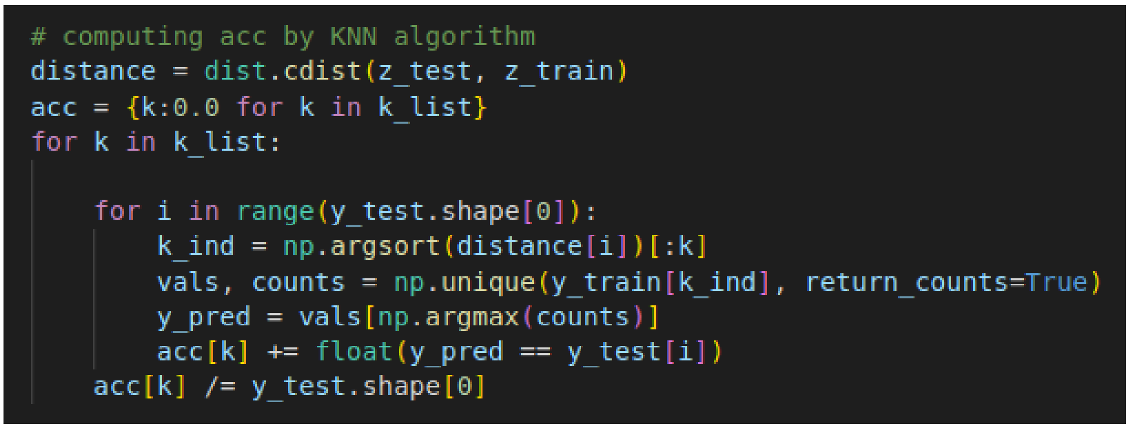


Face reconstruction for random 10 images:
(Top : true faces, bottom : reconstructed faces)



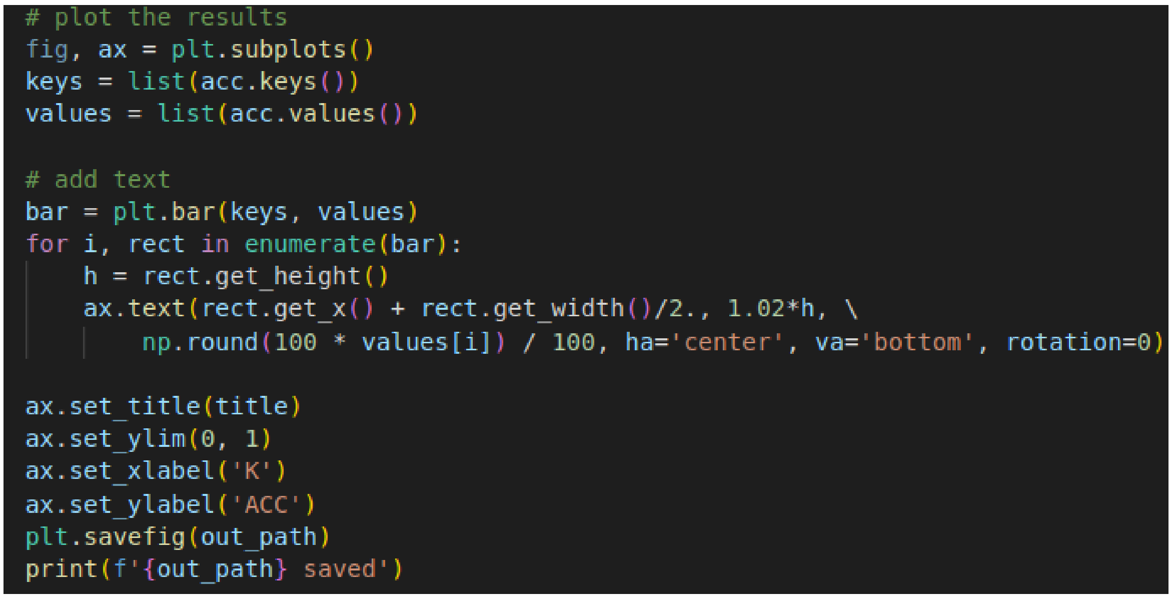
2. Face recogntion

This part we will use KNN algorithm to judge that each face belongs to which subject. The KNN implementation is shown below. Here I will use different k to do the prediction (k = 1 ~ 10) and compare the results



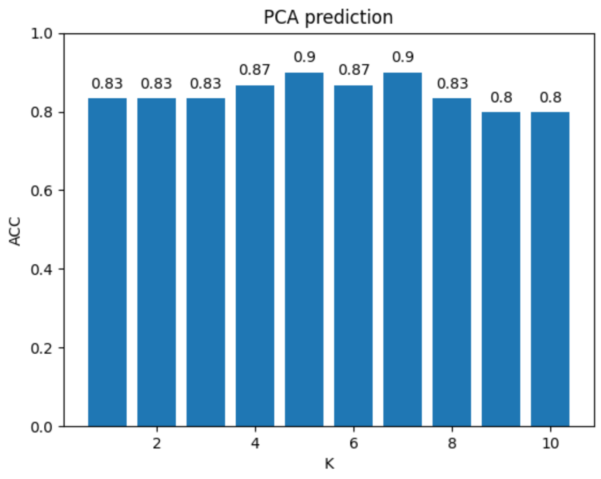
```python
# computing acc by KNN algorithm
distance = dist.cdist(z_test, z_train)
acc = {k:0.0 for k in k_list}
for k in k_list:

    for i in range(y_test.shape[0]):
        k_ind = np.argsort(distance[i])[:k]
        vals, counts = np.unique(y_train[k_ind], return_counts=True)
        y_pred = vals[np.argmax(counts)]
        acc[k] += float(y_pred == y_test[i])
    acc[k] /= y_test.shape[0]
```

Next I will use bar chart to visualize the prediction result accuracy for each k

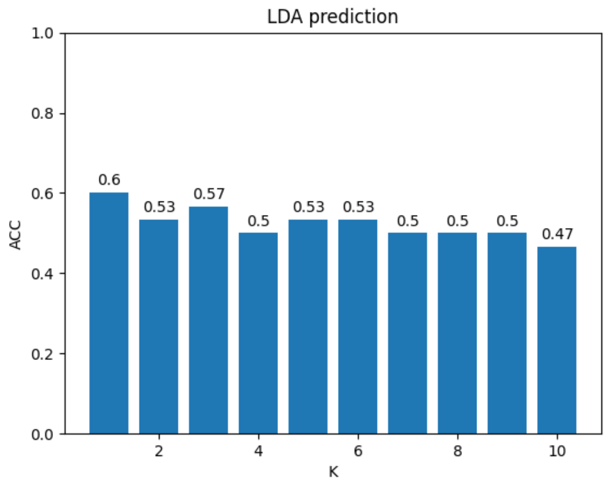


```python
# plot the results
fig, ax = plt.subplots()
keys = list(acc.keys())
values = list(acc.values())

# add text
bar = plt.bar(keys, values)
for i, rect in enumerate(bar):
    h = rect.get_height()
    ax.text(rect.get_x() + rect.get_width()/2., 1.02*h, \
        np.round(100 * values[i]) / 100, ha='center', va='bottom', rotation=0)

ax.set_title(title)
ax.set_ylim(0, 1)
ax.set_xlabel('K')
ax.set_ylabel('ACC')
plt.savefig(out_path)
print(f'{out_path} saved')
```

Accuracy of PCA face recognition:



Accuracy of LDA face recognition:



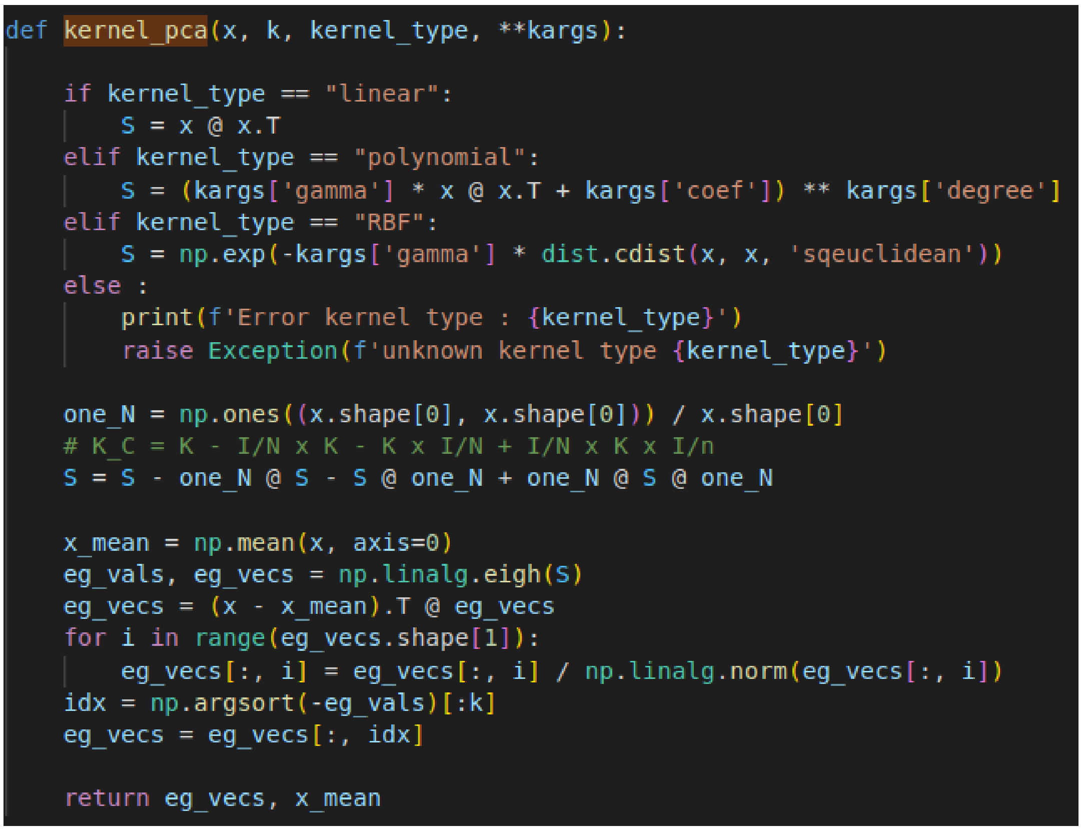
3. Kernel PCA/ Kernel LDA

In my implementation, I will use linear kernel, polynomial kernel with degree 3 and gamma 0.001 and RBF kernel with gamma 0.001.

For kernel PCA, an important thing is that after computing the gram matrix of the original data, we need to do the following mapping to the final matrix that we will use it to compute the eigenvectors:

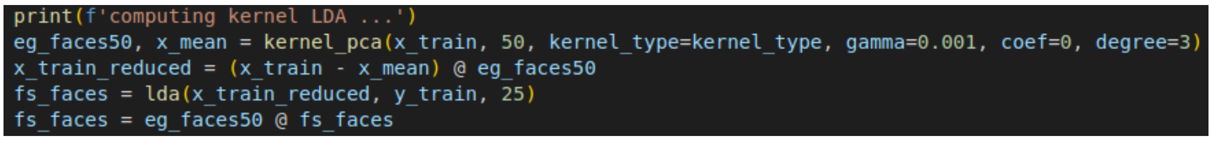
$$K^C = K - \mathbf{1}_N K - K\mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$$

The code is shown below :



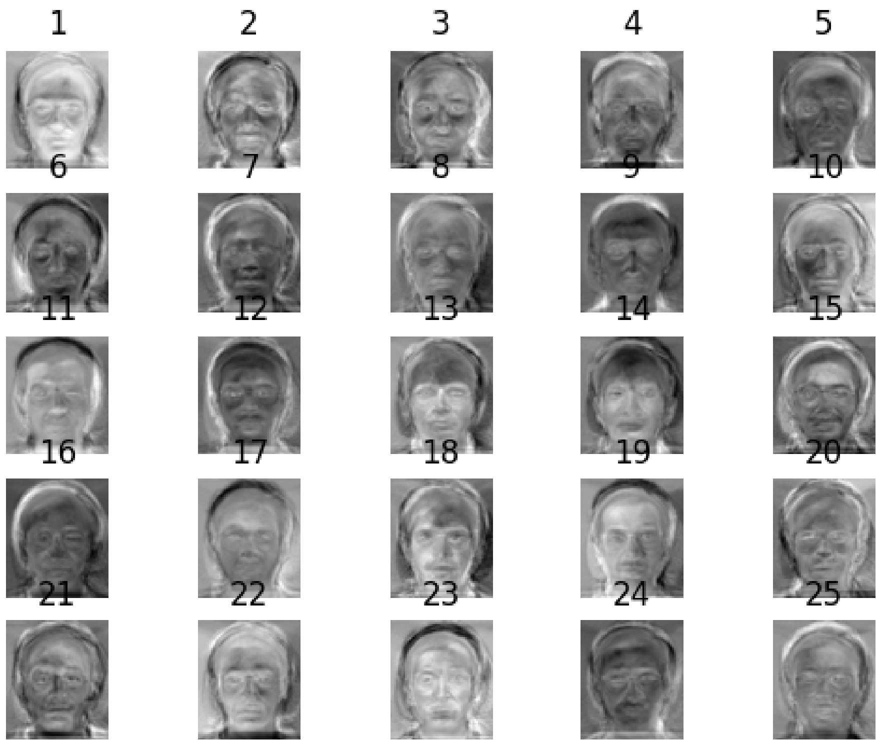
```python
def kernel_pca(x, k, kernel_type, **kargs):

    if kernel_type == "linear":
        S = x @ x.T
    elif kernel_type == "polynomial":
        S = (kargs['gamma'] * x @ x.T + kargs['coef']) ** kargs['degree']
    elif kernel_type == "RBF":
        S = np.exp(-kargs['gamma'] * dist.cdist(x, x, 'sqeuclidean'))
    else :
        print(f'Error kernel type : {kernel_type}')
        raise Exception(f'unknown kernel type {kernel_type}')

    one_N = np.ones((x.shape[0], x.shape[0])) / x.shape[0]
    # K_C = K - I/N x K - K x I/N + I/N x K x I/n
    S = S - one_N @ S - S @ one_N + one_N @ S @ one_N

    x_mean = np.mean(x, axis=0)
    eg_vals, eg_vecs = np.linalg.eigh(S)
    eg_vecs = (x - x_mean).T @ eg_vecs
    for i in range(eg_vecs.shape[1]):
        eg_vecs[:, i] = eg_vecs[:, i] / np.linalg.norm(eg_vecs[:, i])
    idx = np.argsort(-eg_vals)[:k]
    eg_vecs = eg_vecs[:, idx]

    return eg_vecs, x_mean
```

For kernel LDA, like mentioned in the previous section, I will use the dimension reduction result as the input of PCA (from W*H to 50), so the kernel type of LDA is followed by kernel PCA. Actually, the LDA function is the same as the original LDA, the only difference is the input comes from the kernel PCA.
See the code snippet below:

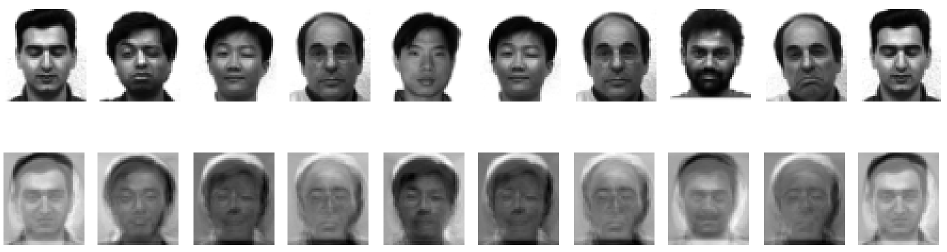


```python
print(f'computing kernel LDA ...')
eg_faces50, x_mean = kernel_pca(x_train, 50, kernel_type=kernel_type, gamma=0.001, coef=0, degree=3)
x_train_reduced = (x_train - x_mean) @ eg_faces50
fs_faces = lda(x_train_reduced, y_train, 25)
fs_faces = eg_faces50 @ fs_faces
```

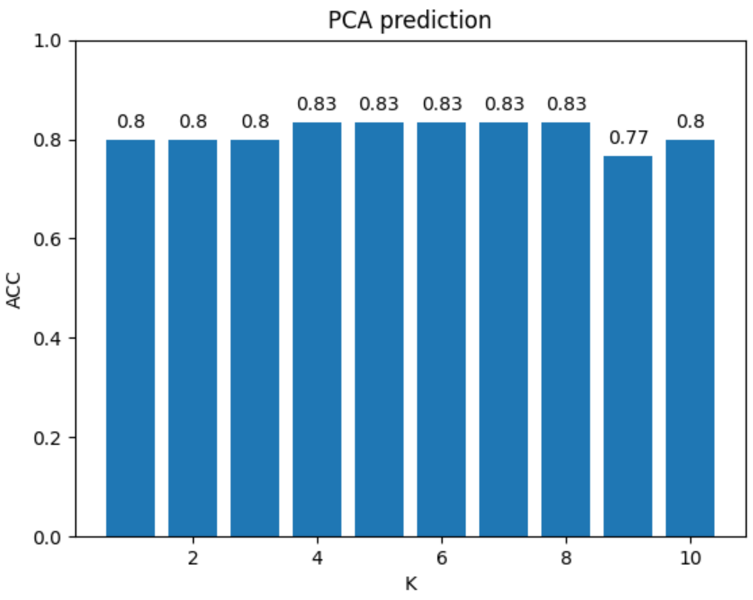
First 25 eigenfaces and 10 face reconstructions by Kernel PCA(Linear):



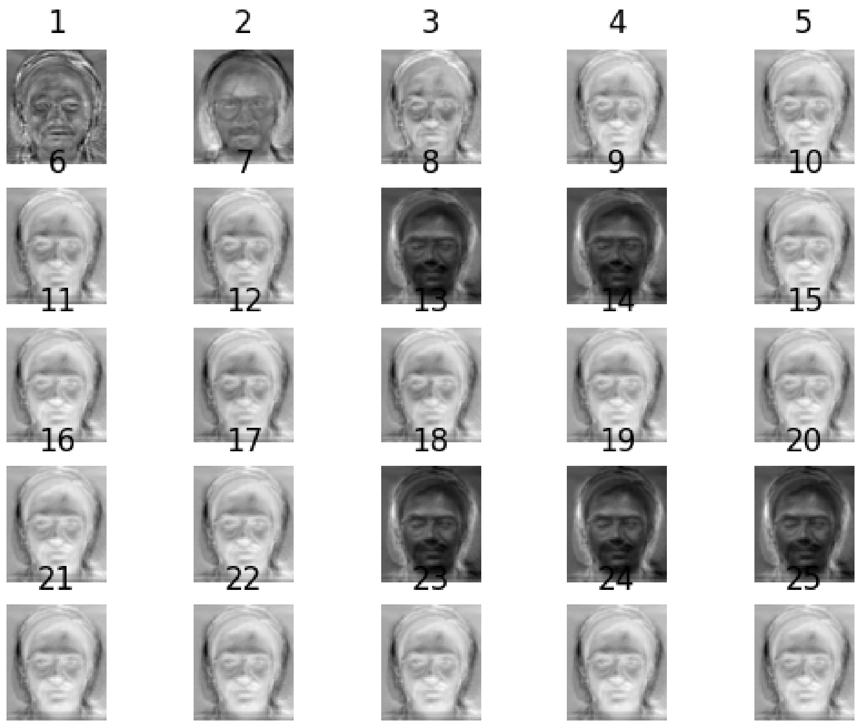
(Top : true faces, bottom : reconstructed faces)



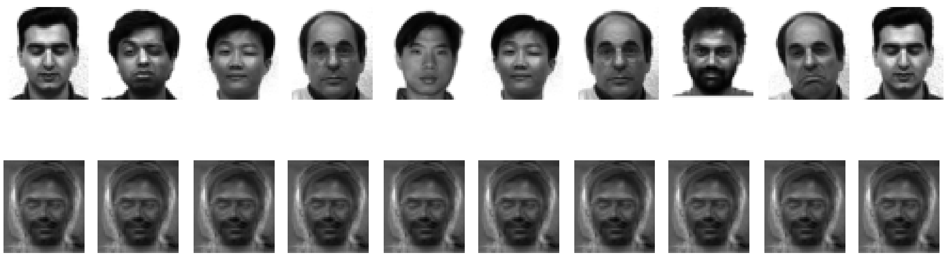
Face reconstruction by Kernel PCA (Linear):



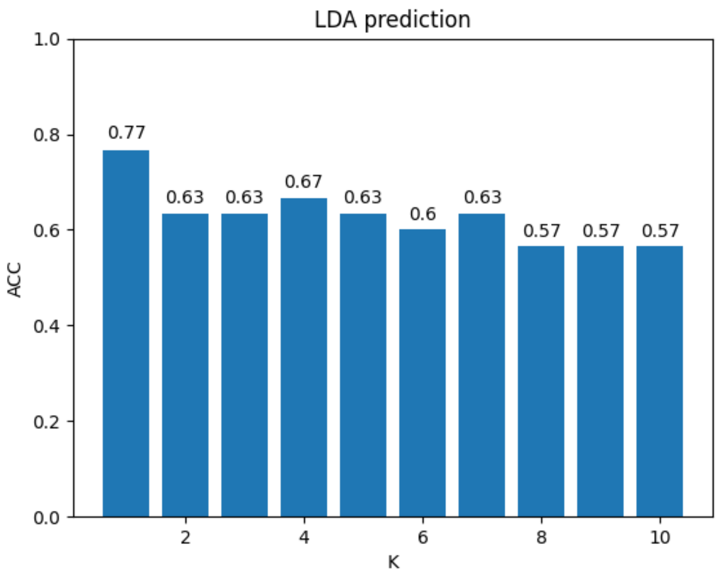
First 25 fisherfaces and 10 face reconstructions by Kernel LDA(Linear):



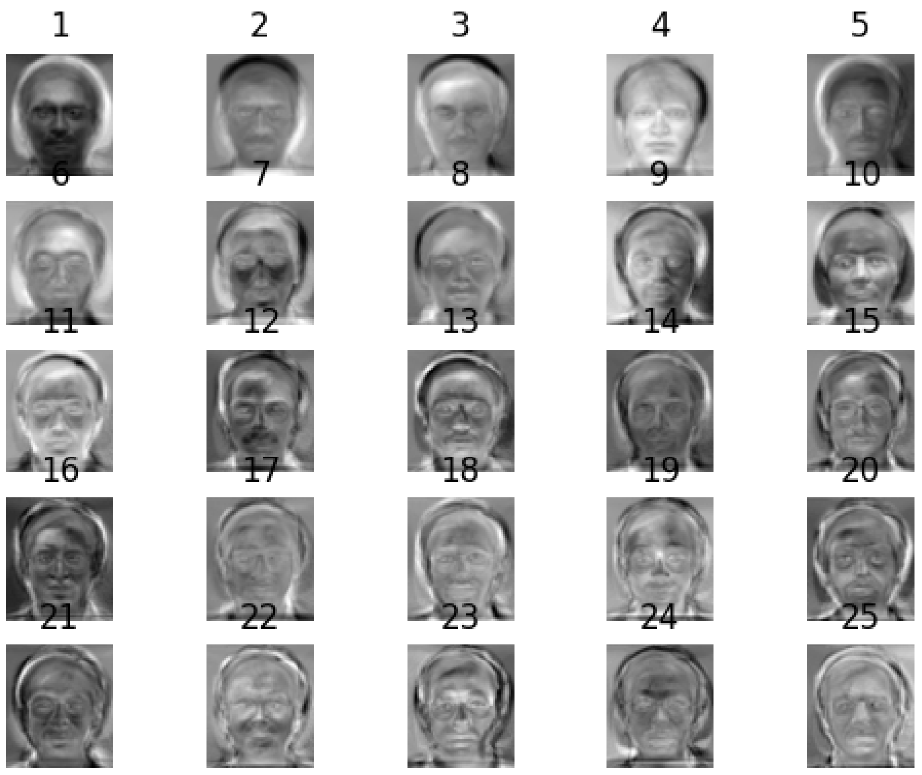
(Top : true faces, bottom : reconstructed faces)



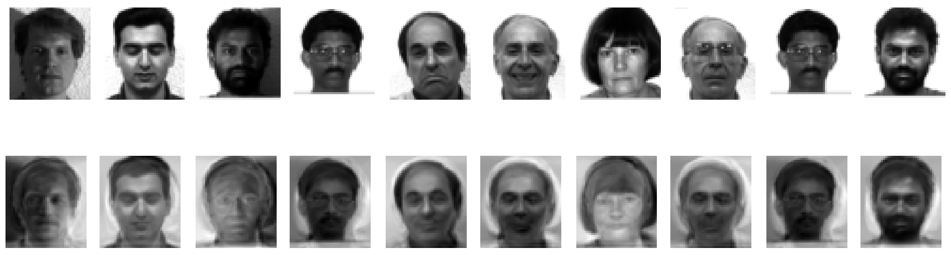
Face reconstruction by Kernel LDA (Linear):



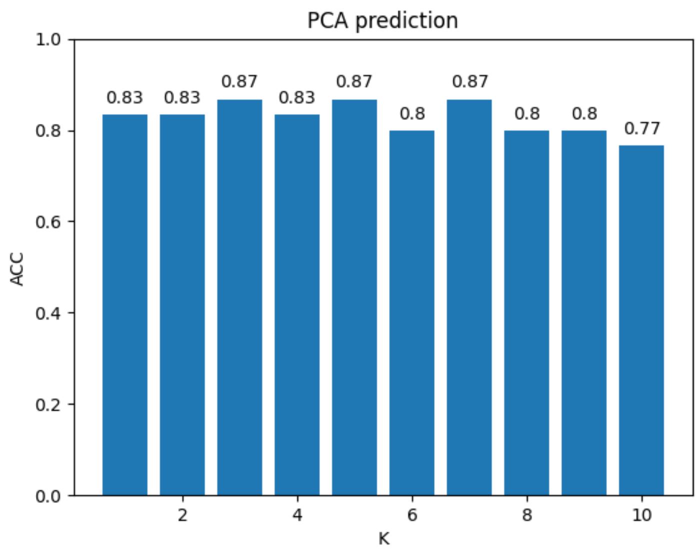
First 25 eigenfaces and 10 face reconstructions by Kernel PCA(Polynomial):



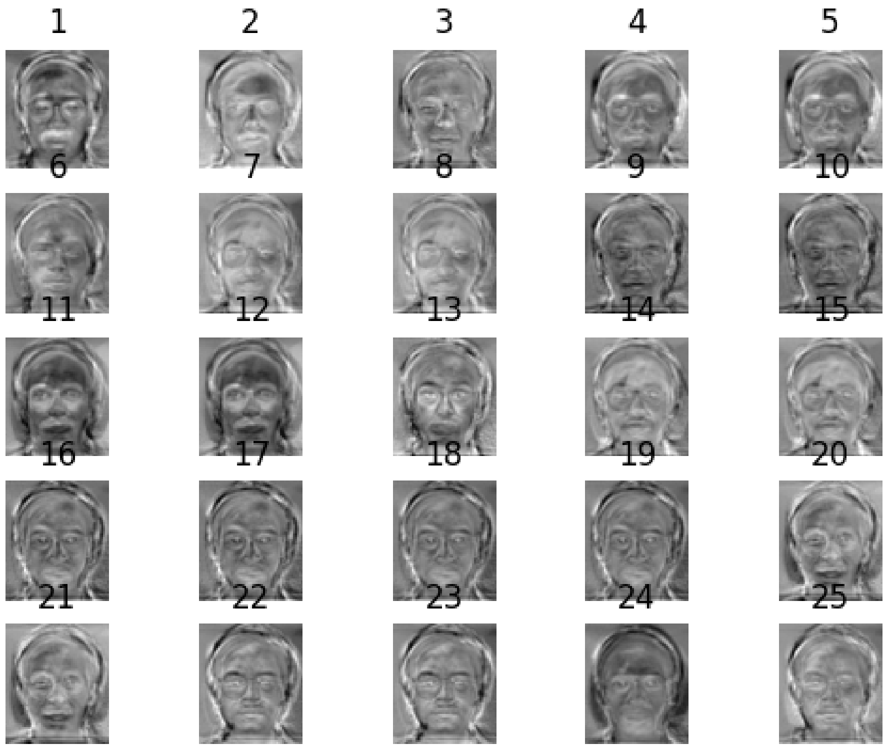
(Top : true faces, bottom : reconstructed faces)



Face reconstruction by Kernel PCA (Polynomial):

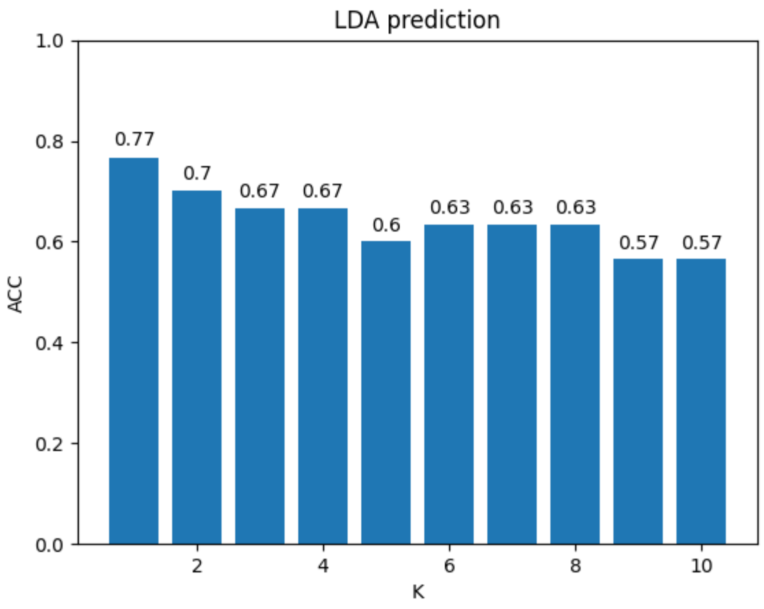


First 25 fisherfaces and 10 face reconstructions by Kernel LDA(Polynomial):

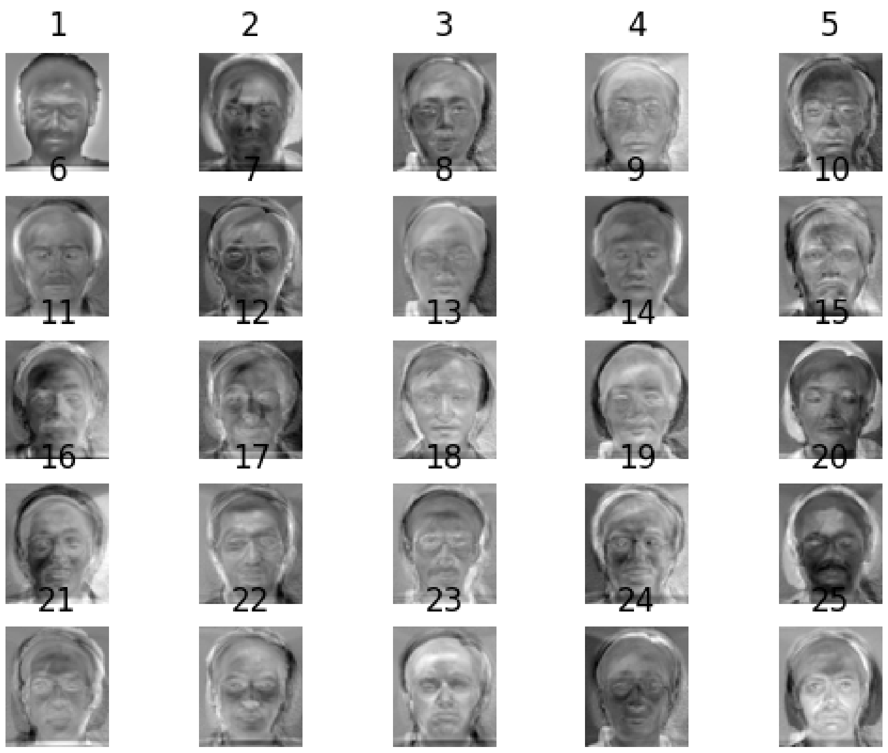


(Top : true faces, bottom : reconstructed faces)

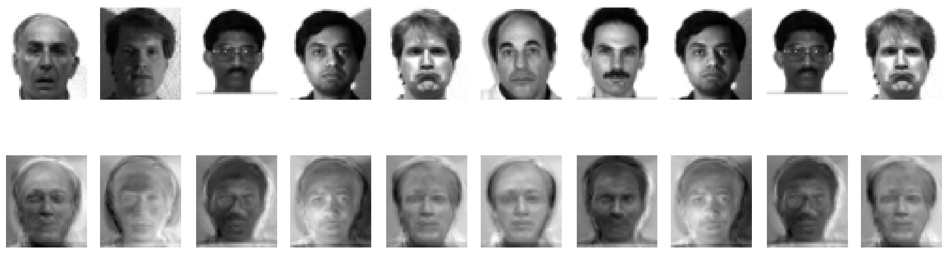
Face reconstruction by Kernel LDA (Polynomial):



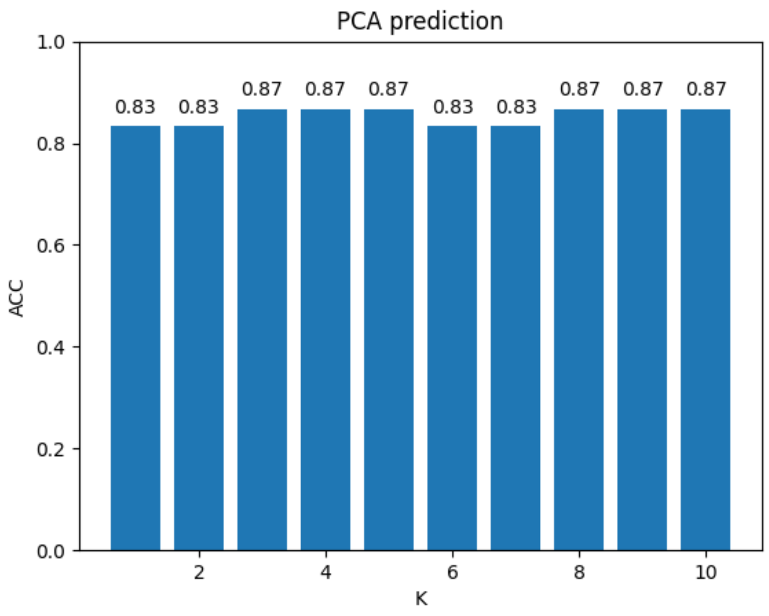
First 25 eigenfaces and 10 face reconstructions by Kernel PCA(RBF):



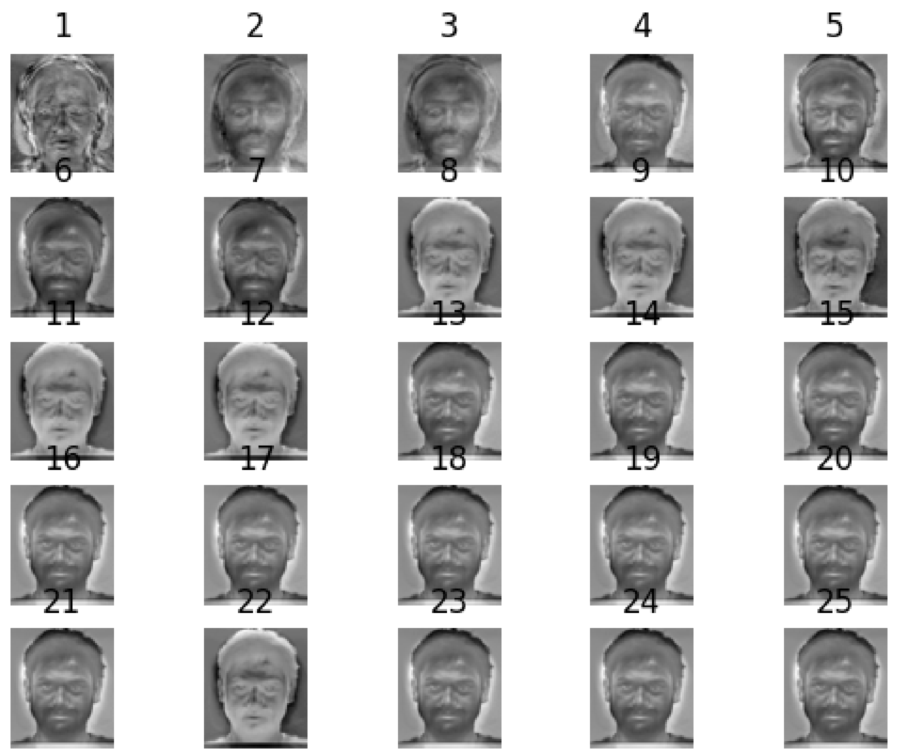
(Top : true faces, bottom : reconstructed faces)



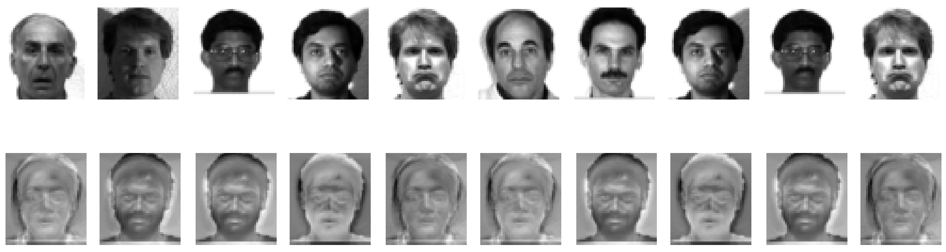
Face reconstruction by Kernel PCA (RBF):



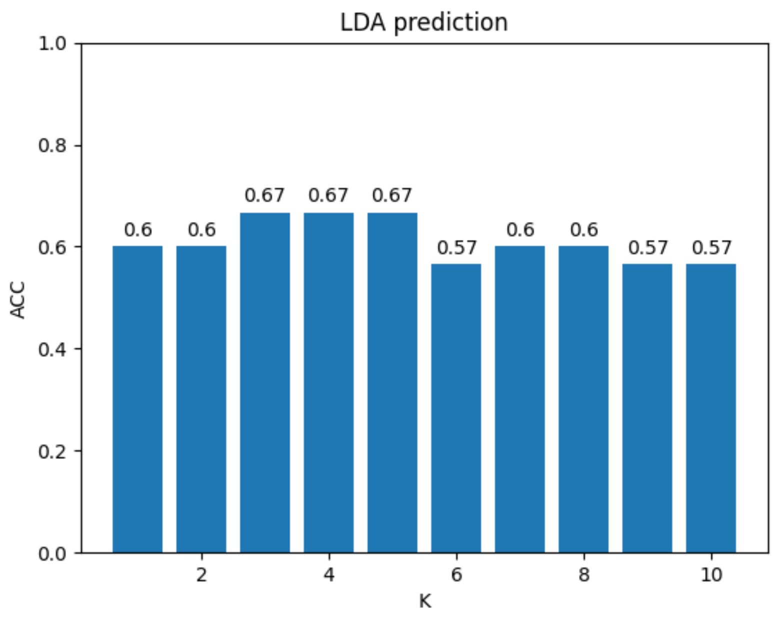
First 25 fisherfaces and 10 face reconstructions by Kernel LDA(RBF):



(Top : true faces, bottom : reconstructed faces)



Face reconstruction by Kernel LDA (RBF):

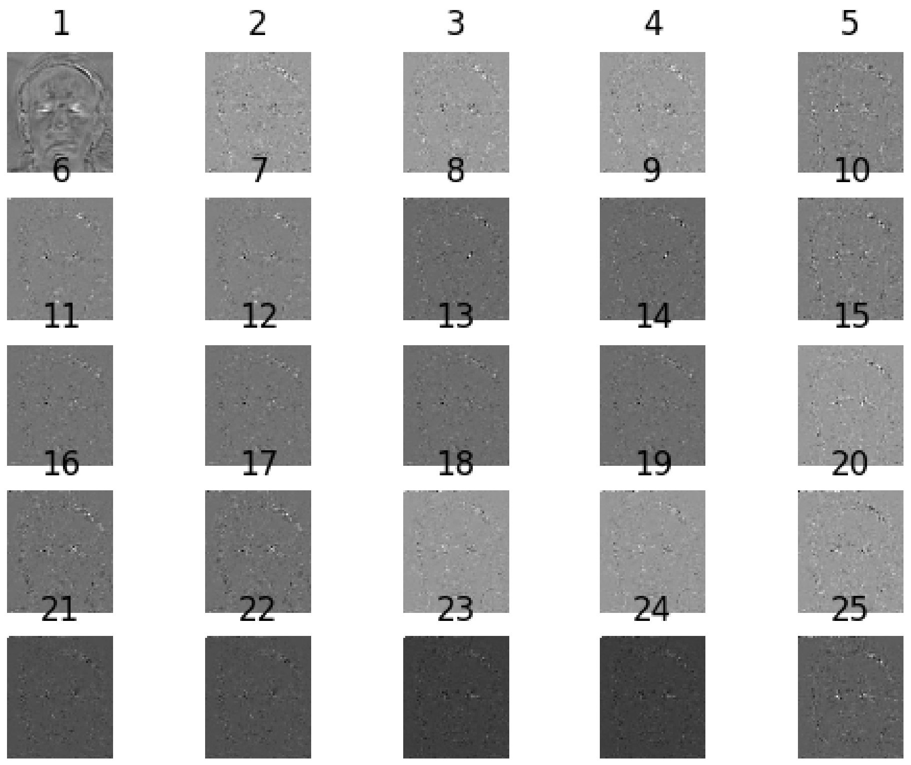


4. Discussion

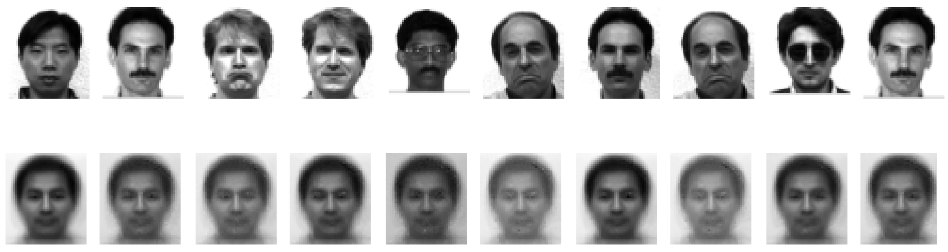
a. PCA and LDA

For PCA, the second implementation mentioned in the previous section is a good choice. Because no matter what resolution I chose, the matrix to compute the eigenvectors is in a constant size (135 * 135), which 135 is the image numbers. As a result, computing the eigenfaces is quite efficient.

For LDA, I also try to input the original data instead of the output of PCA, the top 25 fisherfaces and the face recognition results are shown below:



(Top : true faces, bottom : reconstructed faces)



We can see the top 25 fisherfaces are quite noisy that only the first fisherface can be recognized as a human face. The face reconstruction results seem hard to distinguish the faces belonging to which subject.

b. Face recognition

From the results image shown in the previous section about face recognition, I discovered that PCA outperforms LDA by using KNN as the prediction method. PCA prediction results are about 0.8 in accuracy, while LDA are about 0.6 in accuracy. Besides, k in 1~10 doesn't affect the PCA results a lot, while LDA performed the best when k is the smallest (k = 1). The larger k would lead to worse prediction results in LDA.

c. Kernl PCA, Kernel LDA

On Average, different kernels in PCA don't perform much differently. Besides, I found that the original PCA performed no worse than the kernel PCA. In contrast, linear kernel or polynomial kernel for LDA seemed better than the RBF kernel according to the prediction accuracy. Furthermore, I also found that kernel LDA outperformed LDA a lot on the prediction accuracy. The prediction accuracies of kernel LDA are averagely 0.1 higher than the original LDA.

Actually, I also tried another way to implement the kernel LDA. I read this (https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis) to get the formula of the kernel LDA, which we need to compute two matrix M, N :

N matrix :

$$N = \sum_{j=1}^{c} \mathbf{K}_j (\mathbf{I} - \mathbf{1}_{l_j}) \mathbf{K}_j^{\mathrm{T}}$$

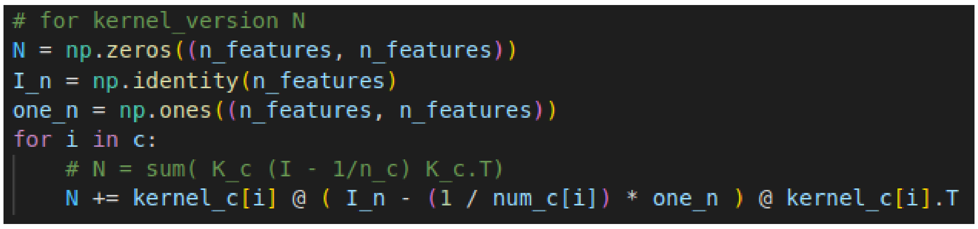
M matrix :

$$M = \sum_{j=1}^{c} l_j (\mathbf{M}_j - \mathbf{M}_*)(\mathbf{M}_j - \mathbf{M}_*)^{\mathrm{T}}$$

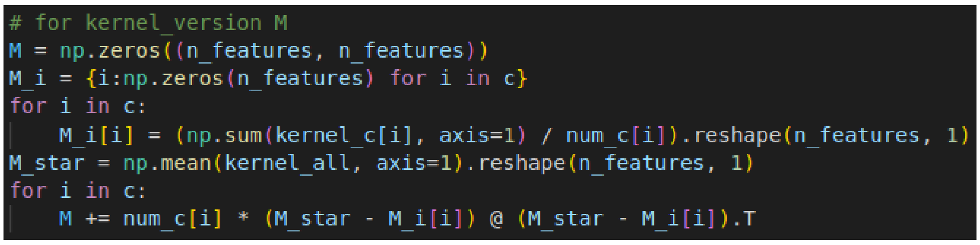
where

$$(\mathbf{M}_*)_j = \frac{1}{l} \sum_{k=1}^{l} k(\mathbf{x}_j, \mathbf{x}_k) \qquad (\mathbf{M}_i)_j = \frac{1}{l_i} \sum_{k=1}^{l_i} k(\mathbf{x}_j, \mathbf{x}_k^i).$$

and the target eigenvectors can be computed from the matrix : $\mathbf{N}^{-1}\mathbf{M}$
The implementation code :



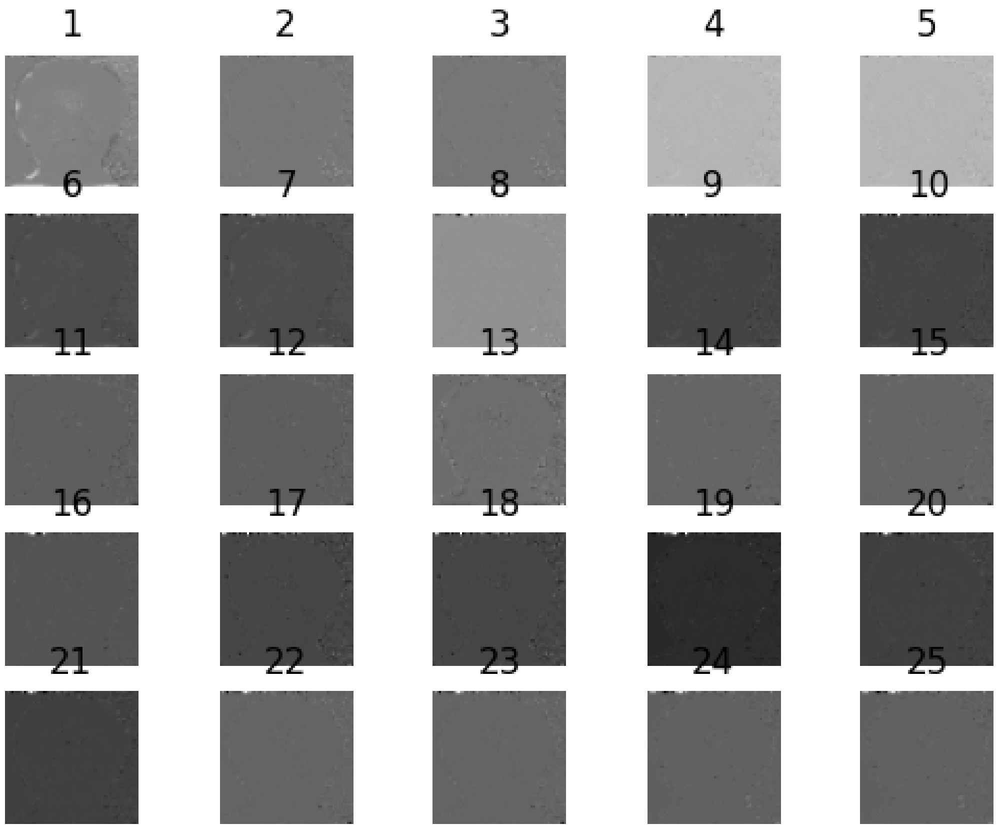
```
# for kernel_version N
N = np.zeros((n_features, n_features))
I_n = np.identity(n_features)
one_n = np.ones((n_features, n_features))
for i in c:
    # N = sum( K_c (I - 1/n_c) K_c.T)
    N += kernel_c[i] @ ( I_n - (1 / num_c[i]) * one_n ) @ kernel_c[i].T
```



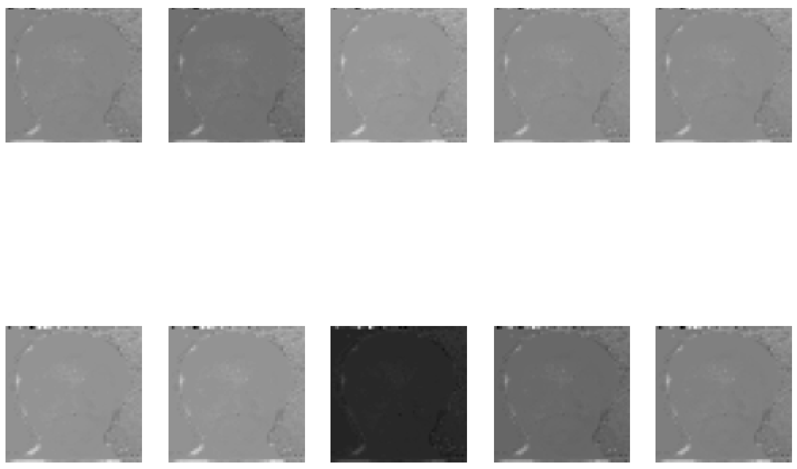
```
# for kernel_version M
M = np.zeros((n_features, n_features))
M_i = {i:np.zeros(n_features) for i in c}
for i in c:
    M_i[i] = (np.sum(kernel_c[i], axis=1) / num_c[i]).reshape(n_features, 1)
M_star = np.mean(kernel_all, axis=1).reshape(n_features, 1)
for i in c:
    M += num_c[i] * (M_star - M_i[i]) @ (M_star - M_i[i]).T
```

The results of this implementation is like what I mentioned previously : fisherfaces will be quite noisy (see example below), I's not sure is this result correct or not, maybe I mislead the meaning of the wiki page.

25 fisherfaces by LDA with polynomial kernel:



10 random subjects' face reconstruction:



II. t-SNE

Code and Experiment Results

1. **Code modification for symmetric SNE**
   There are two things we need to modify in the example t-SNE python code:

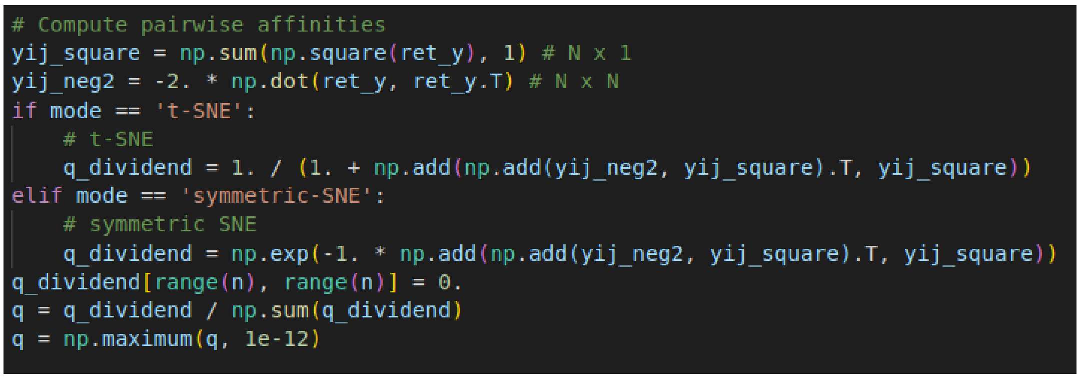
   (1) the low dimensional distribution function q.

   in t-SNE :

$$q_{ij} = \frac{(1+ \| y_i - y_j \|^2)^{-1}}{\sum_{k \neq l}(1+ \| y_i - y_j \|^2)^{-1}}$$

   in symmetric SNE

$$q_{ij} = \frac{\exp(- \| y_i - y_j \|^2)}{\sum_{k \neq l} \exp(- \| y_l - y_k \|^2)}$$



```
# Compute pairwise affinities
yij_square = np.sum(np.square(ret_y), 1) # N x 1
yij_neg2 = -2. * np.dot(ret_y, ret_y.T) # N x N
if mode == 't-SNE':
    # t-SNE
    q_dividend = 1. / (1. + np.add(np.add(yij_neg2, yij_square).T, yij_square))
elif mode == 'symmetric-SNE':
    # symmetric SNE
    q_dividend = np.exp(-1. * np.add(np.add(yij_neg2, yij_square).T, yij_square))
q_dividend[range(n), range(n)] = 0.
q = q_dividend / np.sum(q_dividend)
q = np.maximum(q, 1e-12)
```

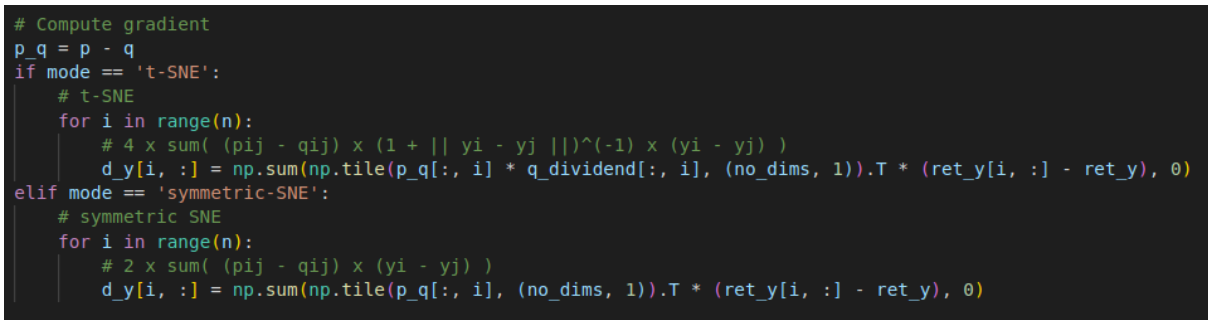
   (2) the gradient of the cost function (KL divergence between p and q).

   in t-SNE :

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1+ \| y_i - y_j \|^2)^{-1}$$

   in symmetric SNE

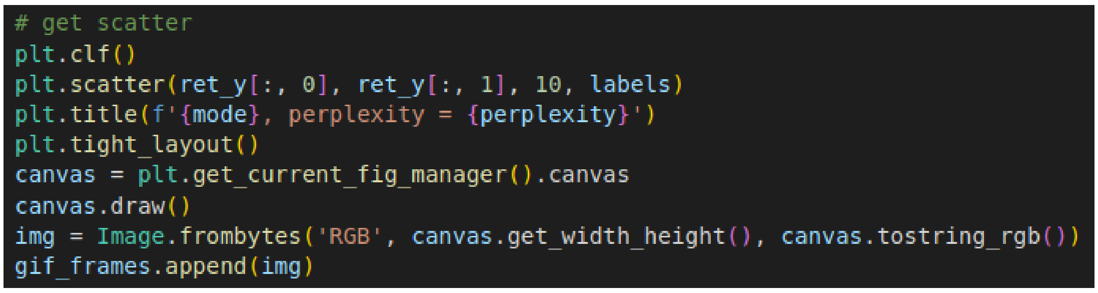
$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$



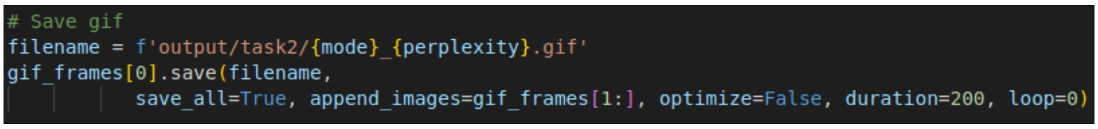
```
# Compute gradient
p_q = p - q
if mode == 't-SNE':
    # t-SNE
    for i in range(n):
        # 4 x sum( (pij - qij) x (1 + || yi - yj ||)^(-1) x (yi - yj) )
        d_y[i, :] = np.sum(np.tile(p_q[:, i] * q_dividend[:, i], (no_dims, 1)).T * (ret_y[i, :] - ret_y), 0)
elif mode == 'symmetric-SNE':
    # symmetric SNE
    for i in range(n):
        # 2 x sum( (pij - qij) x (yi - yj) )
        d_y[i, :] = np.sum(np.tile(p_q[:, i], (no_dims, 1)).T * (ret_y[i, :] - ret_y), 0)
```

2. Visualize the embedding of both t-SNE and symmetric SNE

For visualization, I used the code snippet below to show the two dimensional embedding for both SNE methods. Additionally, I store the image frames per 10 iterations and then save them to the .gif files. Because .gif files cannot shown in pdf format, I will put them in the assignment folder.

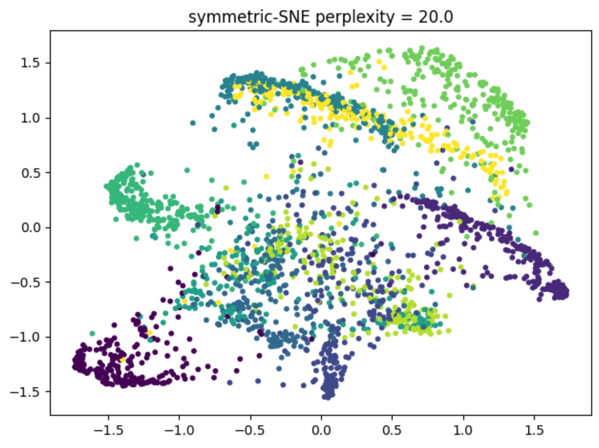


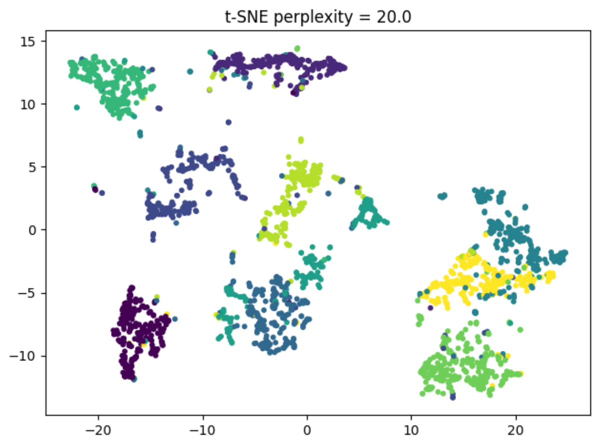
```
# get scatter
plt.clf()
plt.scatter(ret_y[:, 0], ret_y[:, 1], 10, labels)
plt.title(f'{mode}, perplexity = {perplexity}')
plt.tight_layout()
canvas = plt.get_current_fig_manager().canvas
canvas.draw()
img = Image.frombytes('RGB', canvas.get_width_height(), canvas.tostring_rgb())
gif_frames.append(img)
```



```
# Save gif
filename = f'output/task2/{mode}_{perplexity}.gif'
gif_frames[0].save(filename,
            save_all=True, append_images=gif_frames[1:], optimize=False, duration=200, loop=0)
```

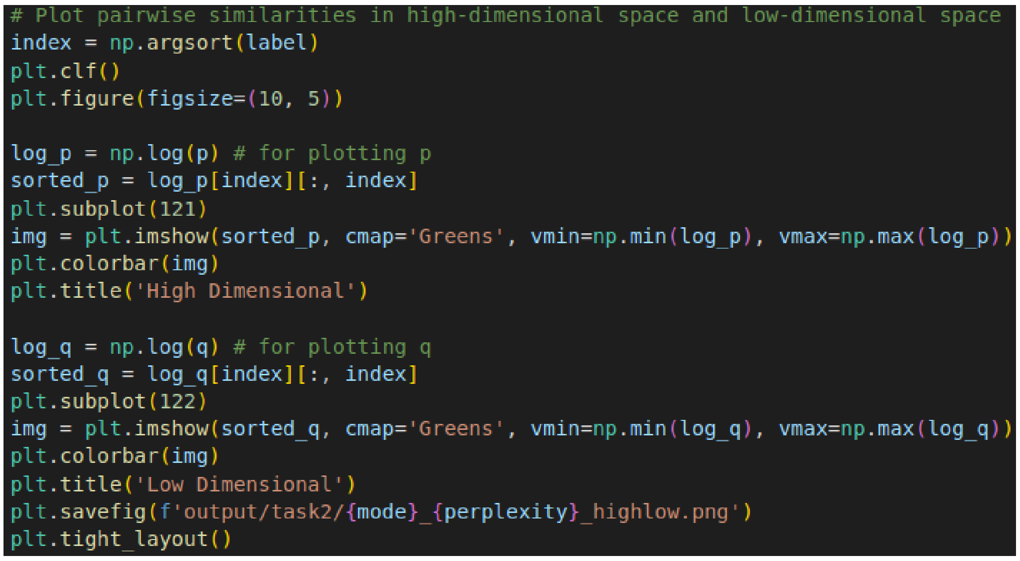
Two dimensional embedding by both symmetric SNE and t-SNE:





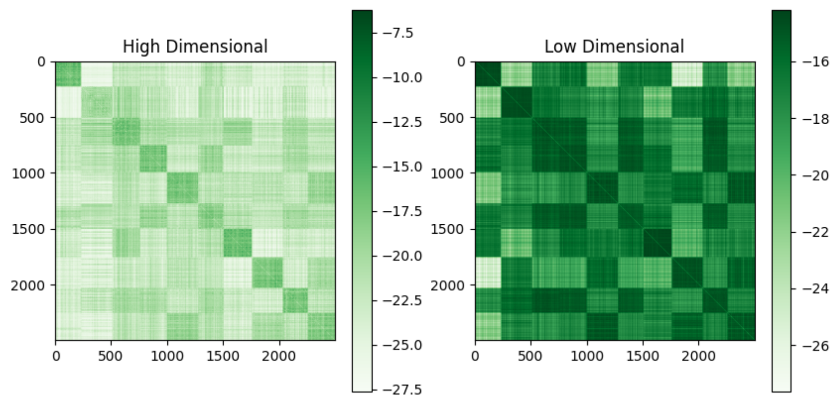
3. Visualize the distribution of pairwise similarities

I used the following code to visualize the pairwise similarities for both high dimensional results and low dimensional results.

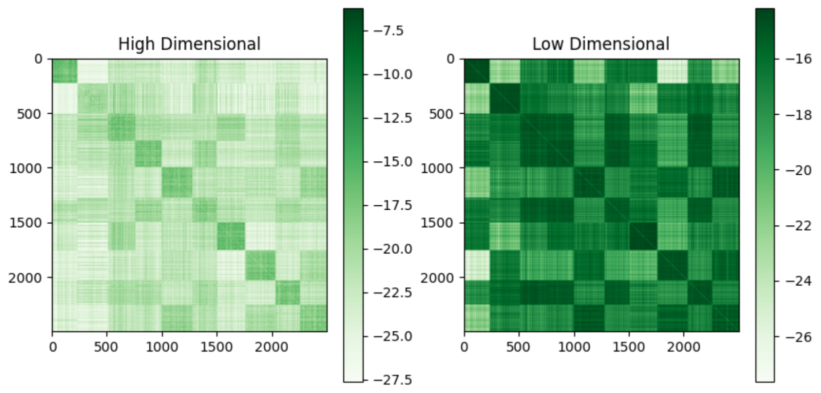


```python
# Plot pairwise similarities in high-dimensional space and low-dimensional space
index = np.argsort(label)
plt.clf()
plt.figure(figsize=(10, 5))

log_p = np.log(p) # for plotting p
sorted_p = log_p[index][:, index]
plt.subplot(121)
img = plt.imshow(sorted_p, cmap='Greens', vmin=np.min(log_p), vmax=np.max(log_p))
plt.colorbar(img)
plt.title('High Dimensional')

log_q = np.log(q) # for plotting q
sorted_q = log_q[index][:, index]
plt.subplot(122)
img = plt.imshow(sorted_q, cmap='Greens', vmin=np.min(log_q), vmax=np.max(log_q))
plt.colorbar(img)
plt.title('Low Dimensional')
plt.savefig(f'output/task2/{mode}_{perplexity}_highlow.png')
plt.tight_layout()
```

High dimension vs Low dimension by symmetric SNE:



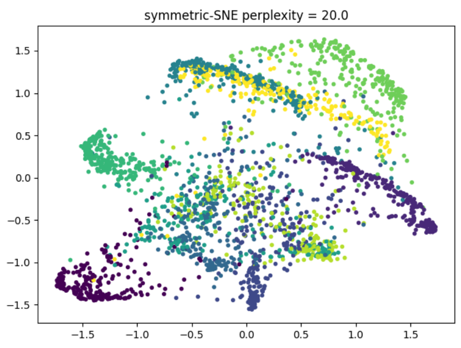
High dimension vs Low dimension by t-SNE:

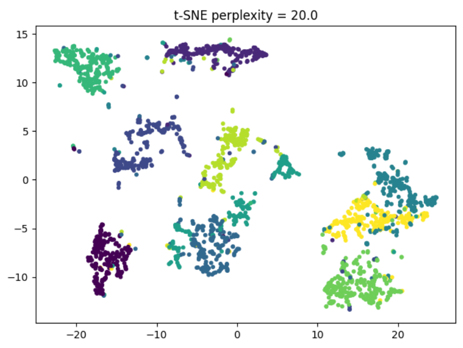


4. Try different perplexity values (PNG)

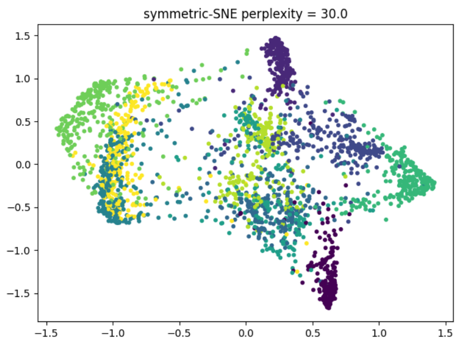
In this section, I applied 20, 30, 40, 50 as the perplexity values.

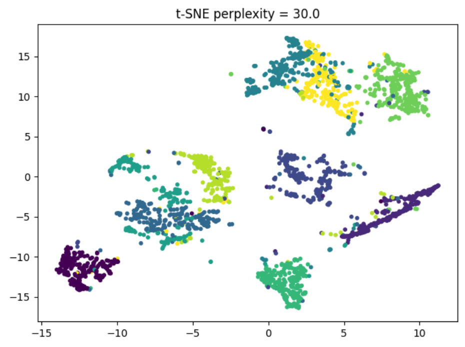
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 20) :



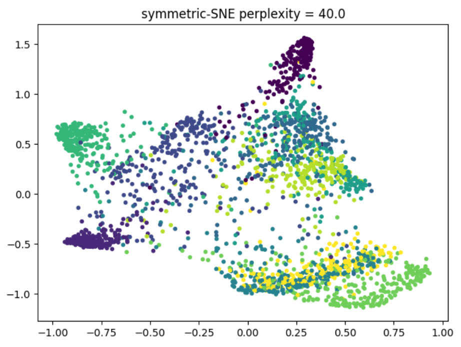


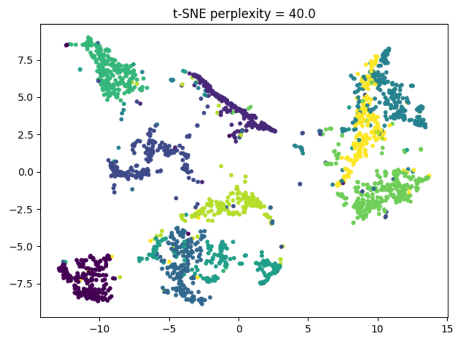
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 30) :



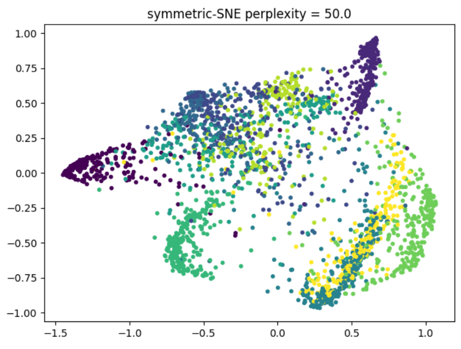


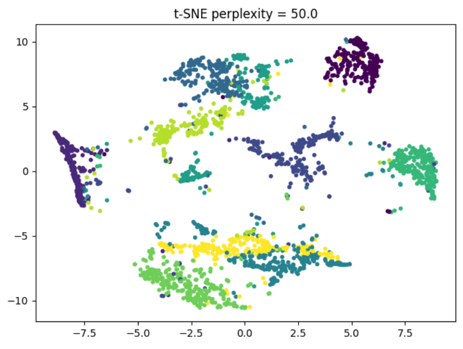
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 40) :





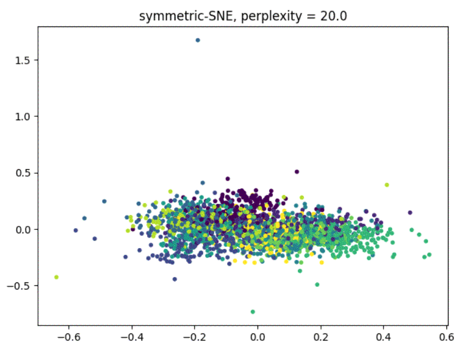
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 50) :

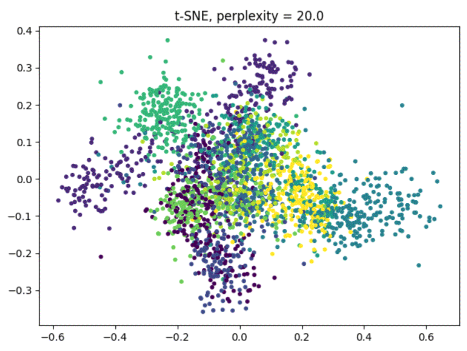




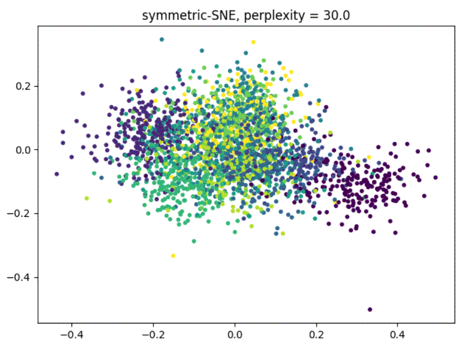
Try different perplexity values (GIF)

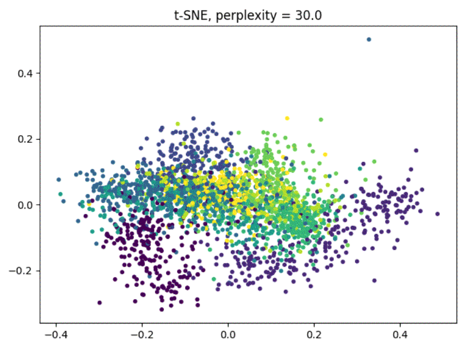
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 20) :



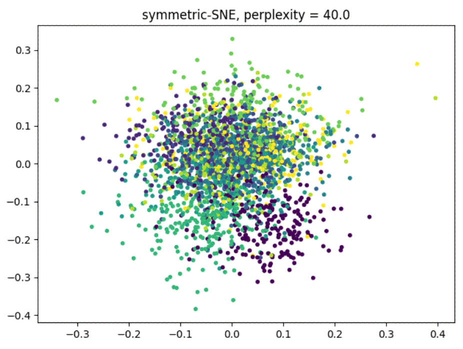


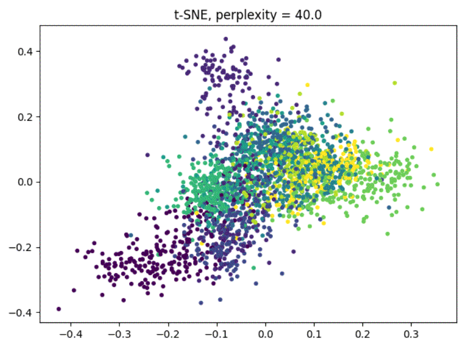
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 30) :



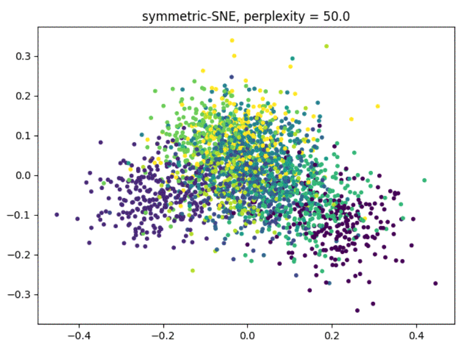


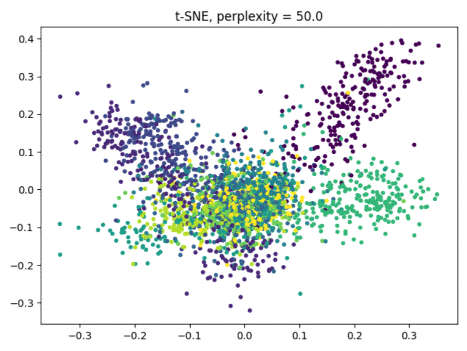
Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 40) :





Two dimensional embedding by symmetric SNE and t-SNE (perplexity = 50) :





5. Discussion

Based on the experimental results shown above, I will list some observations.

From the scatter graphs above, we can learn that symmetric SNE suffers from the crowded problem. On the other hand, t-SNE uses t-distribution to alleviate the crowded problem. Points that are far away from each other in high-dimensional space still have far distances from each other in low-dimensional space.

Perplexity in SNE will affect the number of neighbors to be considered. Theoretically, larger perplexity numbers will make the algorithm less sensitive to the smaller group. Because symmetric SNE suffers from the crowded problem, the effect of perplexity is not clear. However, the effect of perplexity is clear in t-SNE as it gets larger.