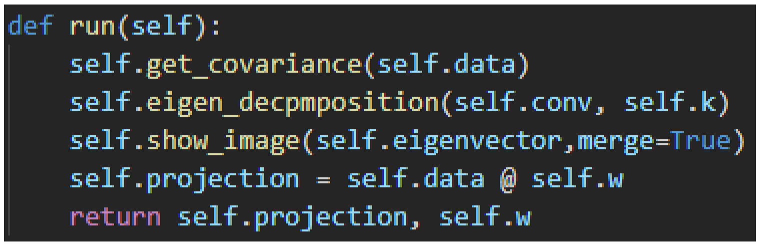
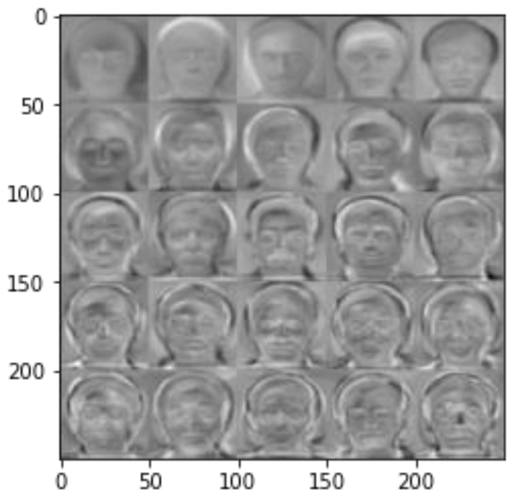
1. Code with detailed explanations:
    - Kernel Eigenfaces
        - Part 1

In the PCA part, our goal is dimensionality reduction by projecting each data point only with the first few principle components to obtain lower dimensional data while preserving as much of the data's variation possible. The first principle component keep the maximum variation and its direction, the second principle component keep the second maximum variation and its direction but also orthogonal to the first component, same story as remain component that we want to keep. And this problem can be solved as the eigenproblem of the covariate matrix of the original data matrix. We find the first k eigenvector which correspond to the first k large eigenvalue. And that k eigenvector is our project matrix, that can let us project origin data point to the dimension reduction space. The following screenshot tell the whole story that how the PCA does.



```
def run(self):
    self.get_covariance(self.data)
    self.eigen_decpmposition(self.conv, self.k)
    self.show_image(self.eigenvector,merge=True)
    self.projection = self.data @ self.w
    return self.projection, self.w
```

And each eigenvector in this problem, can also look as the eigenface, the eigenface which include the most important feature of the origin data space, and we also can visualize it, like the following image, the left top face has the maximum eigenvalue, and the right bottom one has the kth large eigenvalue.



We can also reconstruction the data from low-dimension space to the original dimension space by time the transport of the projection matrix, the equation we be like:

$$low\ dimension\ data = origin\ diemsion\ data\ @\ projection\ matrix$$

$$reconsruction\ data = low\ dimension\ data\ @\ (projection\ matrix)^T$$

The following image show the original data face and reconstruction data face.

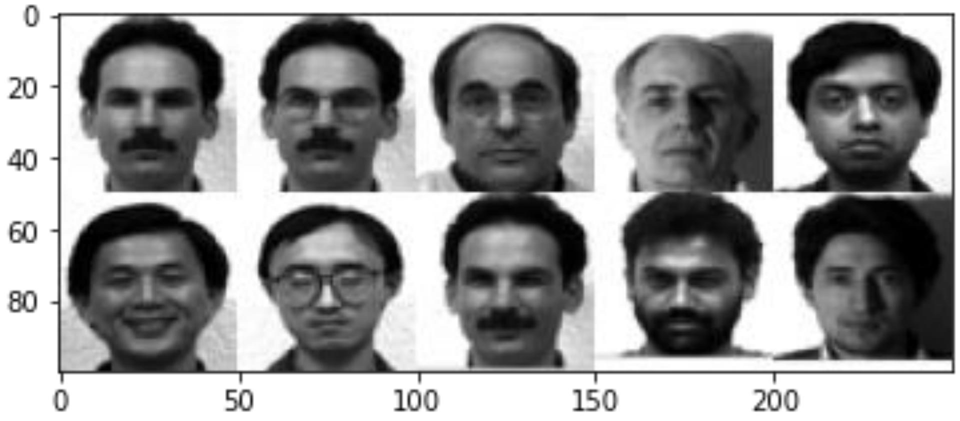


Image 1: original data face

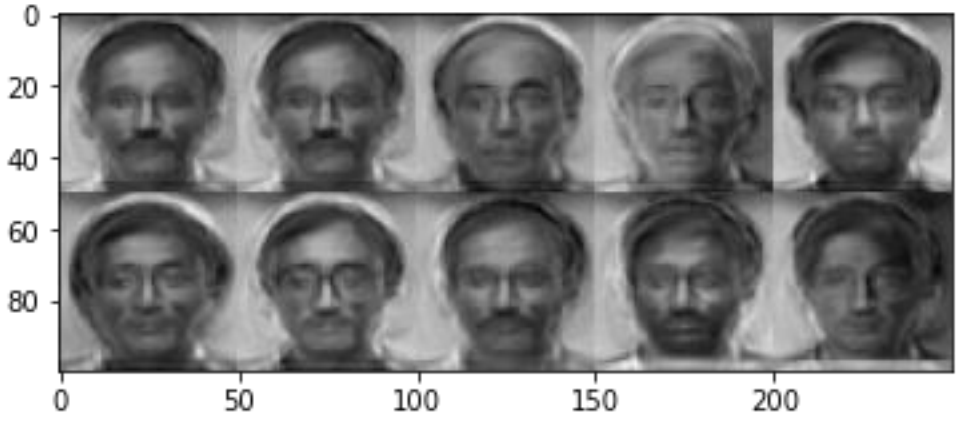
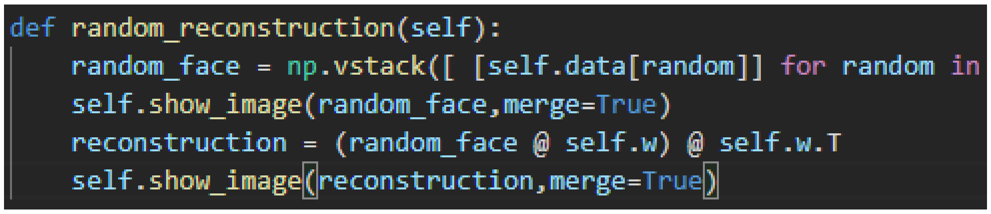


Image 2: reconstruction data face

The following screenshot is the implement of reconstruction.



```
def random_reconstruction(self):
    random_face = np.vstack([ [self.data[random]] for random in
    self.show_image(random_face,merge=True)
    reconstruction = (random_face @ self.w) @ self.w.T
    self.show_image(reconstruction,merge=True)
```

In the LDA part, the goal is also dimensionality reduction and more commonly later classification, but is in the different point of view, in LDA we want to project data to some dimension space, in the simplest case is one-dimensional space also a line, for instance if we have two class of data set, and we want project those data to a line, the data on that line can split data to two set, we also hope that projected data has some property, first is projected data of C1 is maximally separated from projected data of C2, like the following equation.

$$m_j = \frac{1}{n_j}\sum_{i\in C_j} y_i = \sum_{i\in C_j} w^T x_i \quad M_j = \frac{1}{n_j}\sum_{i\in C_j} x_i$$

$$|m_2 - m_1| = |w^T(M_2 - M_1)| \ \textit{is named as between class scatter}$$

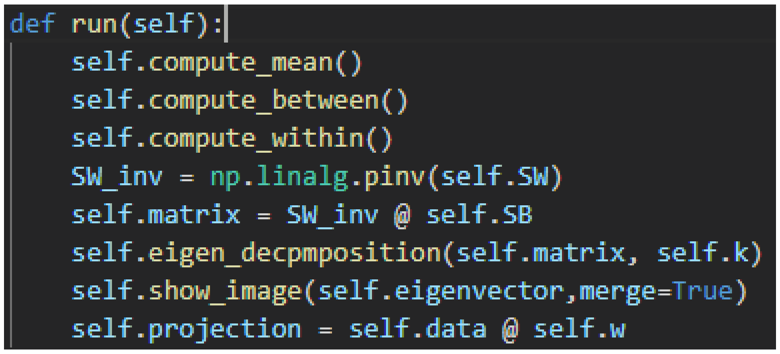
The second property is we want to minimize variance of each projected class, like the following equation.

$$s_j^2 = \sum_{i\in C_j}(y_i - m_j)^2$$

Two property or condition, we want to maximize between-class scatter and minimize within-class scatter, can be written as following objective function:

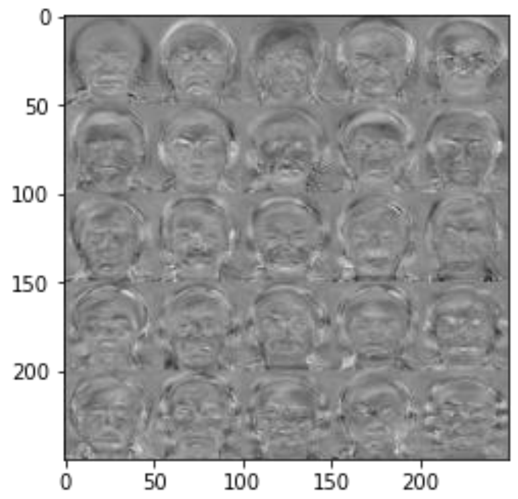
$$J(w) = \frac{(m_2 - m_1)^2}{s_1^2 + s_2^2} = \frac{w^T S_B w}{w^T S_W w}$$

This problem is Rayleigh quotient also can be solved by the eigenproblem, like the same story as PCA, we also want to find the projection matrix w, where the w is composed by the first k eigenvector that corresponds to the first k large eigenvalue. The following screenshot is how I implement the LDA.



```
def run(self):
    self.compute_mean()
    self.compute_between()
    self.compute_within()
    SW_inv = np.linalg.pinv(self.SW)
    self.matrix = SW_inv @ self.SB
    self.eigen_decpmposition(self.matrix, self.k)
    self.show_image(self.eigenvector,merge=True)
    self.projection = self.data @ self.w
```

Remain story is same as PCA, we also have fisherface also can restructure to origin data space, the following image show the result of LDA.



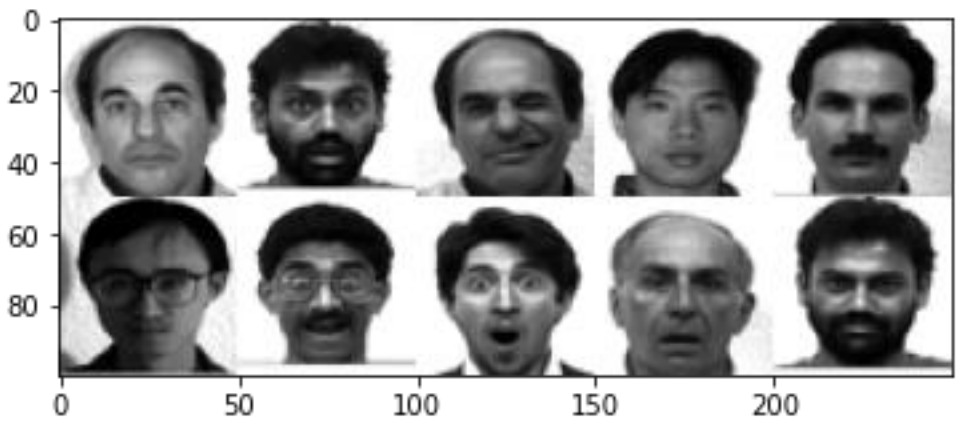


Image 1: original data face

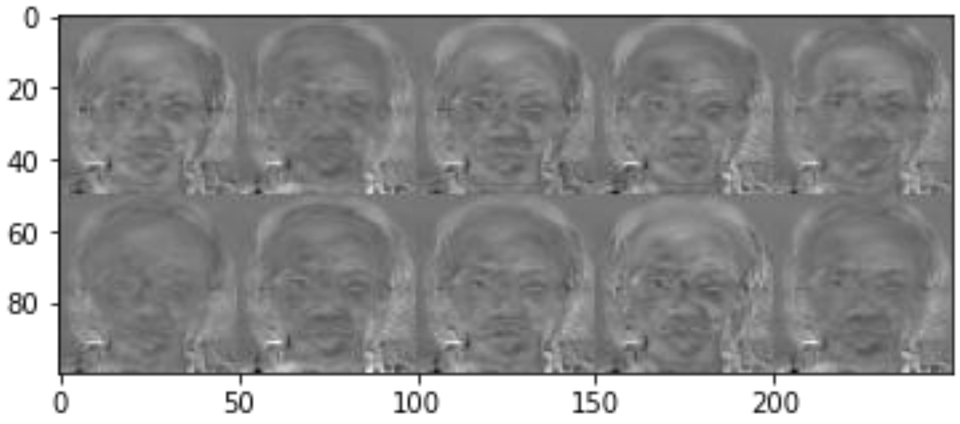


Image 2: reconstruction data face

- Part 2

PCA and LDA can be used as a linear classifier because we can find the feature and most important principal component of the training data, that is eigenvector, the eigenvectors also can be used as the basis of the eigenspace, we can use that basis be the projection matrix. when the testing data coming, project it to the eigenspace as know as low-dimension space, by the projection matrix, now testing data and training data are in the same low-dimension space, so we can use the simplest classification algorithm, like KNN.

KNN is just like the voting, the point finds the most k closest neighbors, and we say the point is same as the most common class.

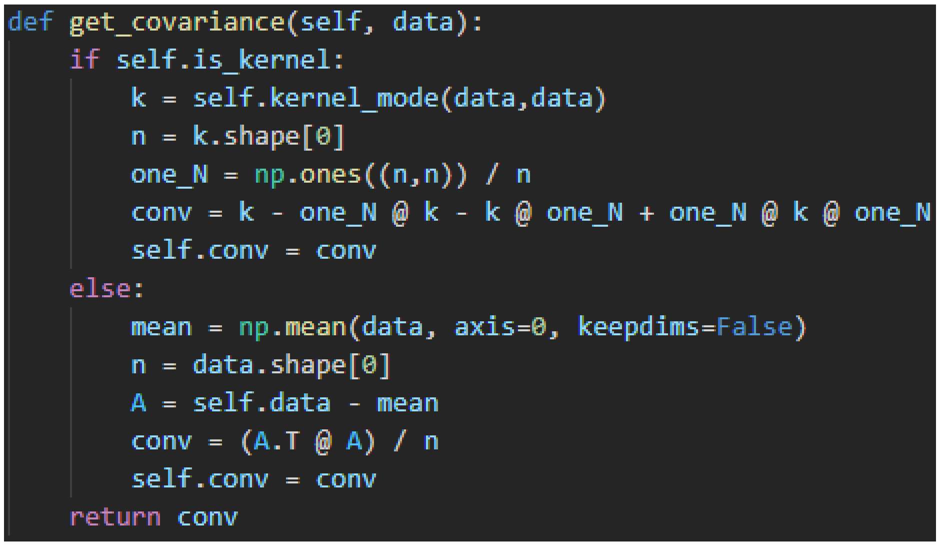
- Part 3

PCA and LDA also can extend to kernel version, for linear version to non-linear version, also use the kernel trick like before. Kernel PCA is easier explain than kernel LDA, we just want to solve the eigenvalue problem of covariance matrix in feature space, like the following equation.

$$\frac{1}{N}\Phi(X)\Phi(X)^T w = \lambda w$$

$$K^C = K - 1_N K - K 1_N + 1_N K 1_N$$

The following screenshot is the main different between simple PCA and kernel PCA, and how I implement.



```
def get_covariance(self, data):
    if self.is_kernel:
        k = self.kernel_mode(data,data)
        n = k.shape[0]
        one_N = np.ones((n,n)) / n
        conv = k - one_N @ k - k @ one_N + one_N @ k @ one_N
        self.conv = conv
    else:
        mean = np.mean(data, axis=0, keepdims=False)
        n = data.shape[0]
        A = self.data - mean
        conv = (A.T @ A) / n
        self.conv = conv
    return conv
```

Still remember that in the LDA we want to maximize between-class scatter and minimize within-class scatter, that is S_B and S_w. In the kernel LDA we want to treat S_B and S_w to the kernel version. The origin data can be mapped to a new feature space, in this feature space our objective function turn into the following equation.

$$J(w) = \frac{w^T S_B^\phi w}{w^T S_W^\phi w}$$

$$w^T S_B^\phi w = w^T \left(m_2^\phi - m_1^\phi\right)\left(m_2^\phi - m_1^\phi\right)^T w = \alpha^T M \alpha \text{ where } M = (M_2 - M_1)(M_2 - M_1)^T$$

$$w^T S_W^\phi w = \alpha^T M \alpha \text{ where } N = \textstyle\sum_{j=1,2} K_j (I - 1_{l_j}) K_j^T$$

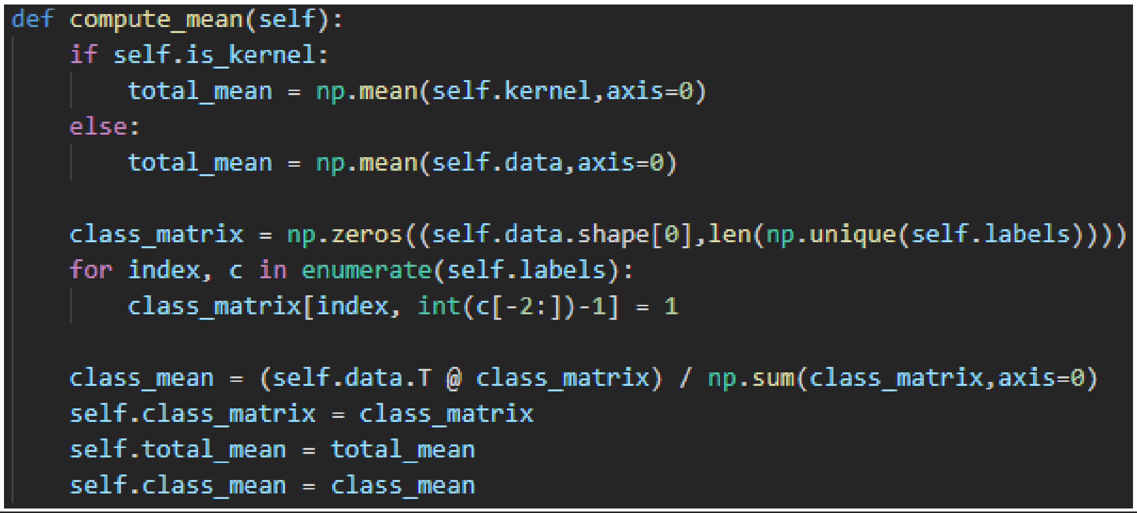
So now equation for objective function can be rewritten as

$$J(\alpha) = \frac{\alpha^T M \alpha}{\alpha^T N \alpha}$$

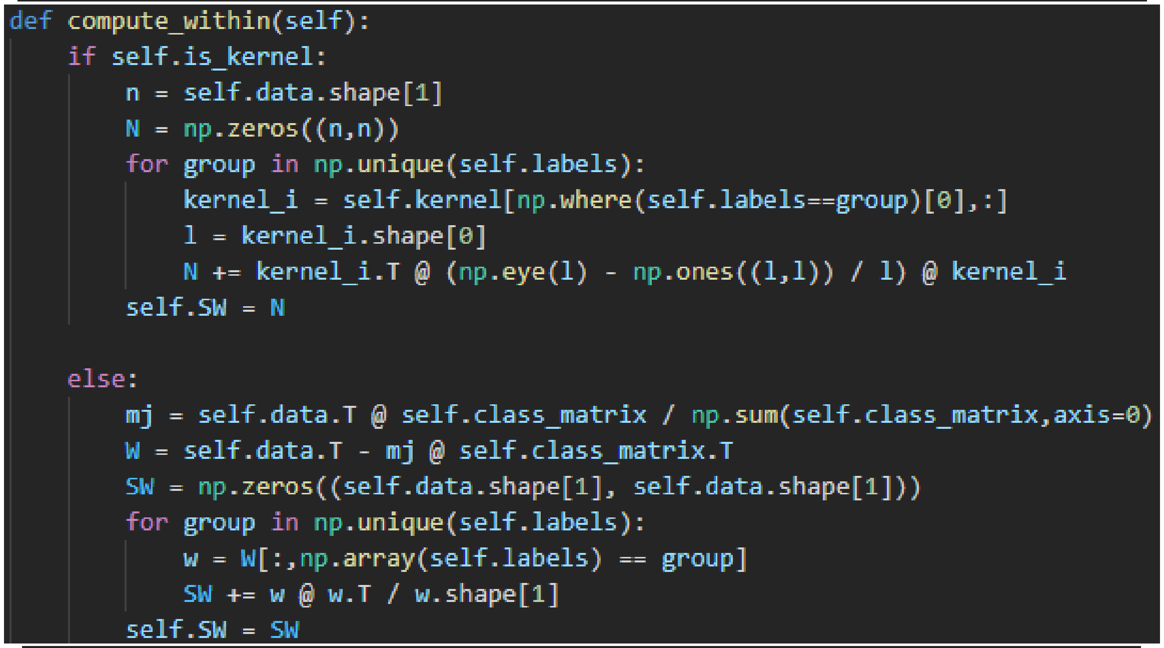
And we the solve α like following equation.

$$\alpha = N^{-1} M$$

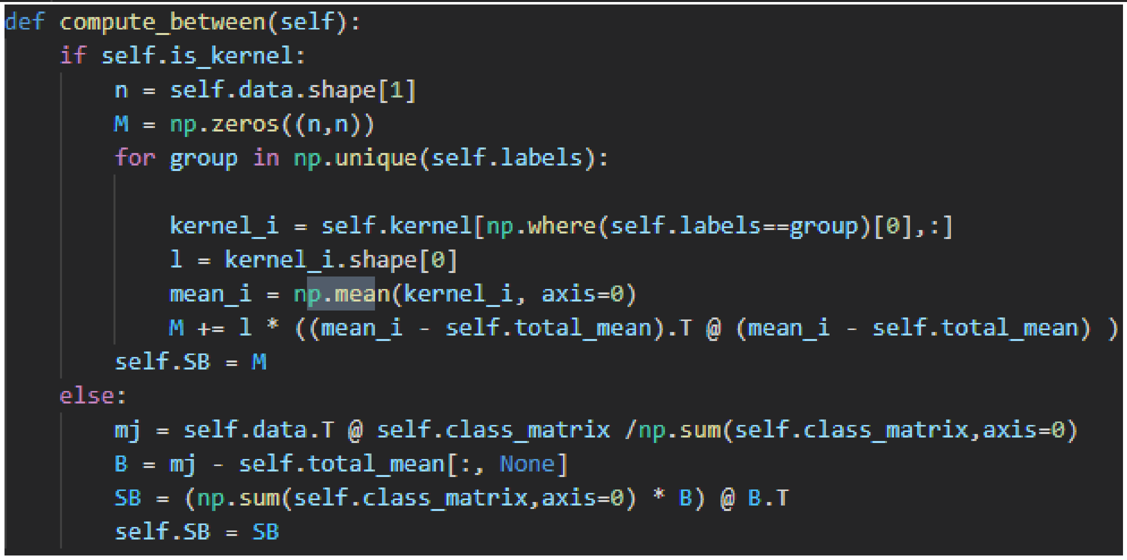
The following few screenshots, are the most important part of the implement of the LDA and the kernel LDA. After compute S_B and S_W and objective matrix $W$and α ,the remain story is like the PCA and kernel PCA, treat $W$and α to the eigenproblem, find the projection matrix that can project the data to high-dimension space to the low-dimension space.



```python
def compute_mean(self):
    if self.is_kernel:
        total_mean = np.mean(self.kernel,axis=0)
    else:
        total_mean = np.mean(self.data,axis=0)

    class_matrix = np.zeros((self.data.shape[0],len(np.unique(self.labels))))
    for index, c in enumerate(self.labels):
        class_matrix[index, int(c[-2:])-1] = 1

    class_mean = (self.data.T @ class_matrix) / np.sum(class_matrix,axis=0)
    self.class_matrix = class_matrix
    self.total_mean = total_mean
    self.class_mean = class_mean
```



```python
def compute_within(self):
    if self.is_kernel:
        n = self.data.shape[1]
        N = np.zeros((n,n))
        for group in np.unique(self.labels):
            kernel_i = self.kernel[np.where(self.labels==group)[0],:]
            l = kernel_i.shape[0]
            N += kernel_i.T @ (np.eye(l) - np.ones((l,l)) / l) @ kernel_i
        self.SW = N

    else:
        mj = self.data.T @ self.class_matrix / np.sum(self.class_matrix,axis=0)
        W = self.data.T - mj @ self.class_matrix.T
        SW = np.zeros((self.data.shape[1], self.data.shape[1]))
        for group in np.unique(self.labels):
            w = W[:,np.array(self.labels) == group]
            SW += w @ w.T / w.shape[1]
        self.SW = SW
```



```python
def compute_between(self):
    if self.is_kernel:
        n = self.data.shape[1]
        M = np.zeros((n,n))
        for group in np.unique(self.labels):

            kernel_i = self.kernel[np.where(self.labels==group)[0],:]
            l = kernel_i.shape[0]
            mean_i = np.mean(kernel_i, axis=0)
            M += l * ((mean_i - self.total_mean).T @ (mean_i - self.total_mean) )
        self.SB = M
    else:
        mj = self.data.T @ self.class_matrix /np.sum(self.class_matrix,axis=0)
        B = mj - self.total_mean[:, None]
        SB = (np.sum(self.class_matrix,axis=0) * B) @ B.T
        self.SB = SB
```

- t-SNE
    - Part 1

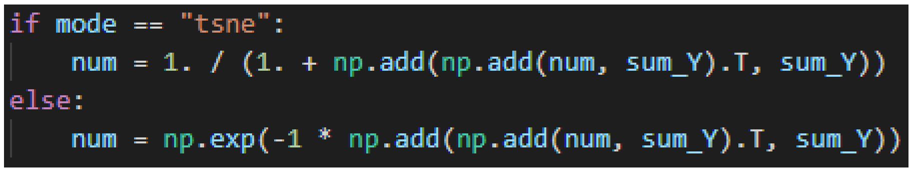
SNE converting the high-dimensional Euclidean distances into conditional probability that represent similarities, and we want to preserve pairwise similarity between high-dimension and low-dimension, used KL divergence to measure the distance between distributions of high-dimension and low-dimension, because the property of KL divergence can be seem as asymmetric, also have some variant like symmetric or t-SNE.

The main different between Symmetric SNE and t-SNE, is the definition of q_{ij} where is the probability represent similarities in the low-dimension space. Both equations are show in the following.

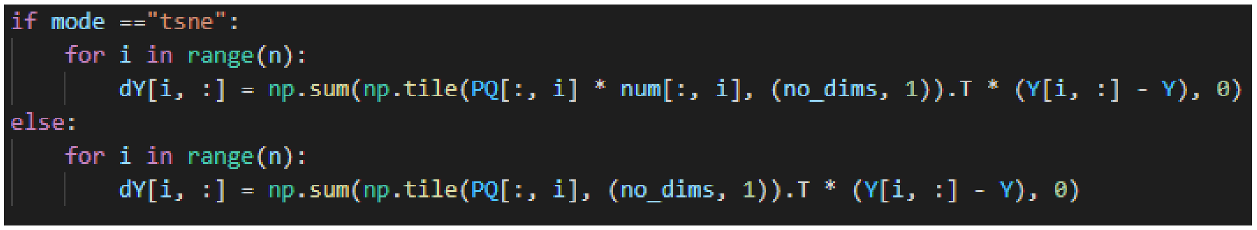
The left-hand side is symmetric SNE, right-hand side is t-SNE.

$$q_{ij} = \frac{\exp\left(-||y_i - y_j||^2\right)}{\sum_{k\neq l} \exp\left(-||y_l - y_k||^2\right)} \quad q_{ij} = \frac{(1+||y_i - y_j||^2)^{-1}}{\sum_{k\neq l}(1+||y_i - y_y||^2)^{-1}}$$

The following screenshot is the implement of the q_{ij} in both SSNE and t-SNE.



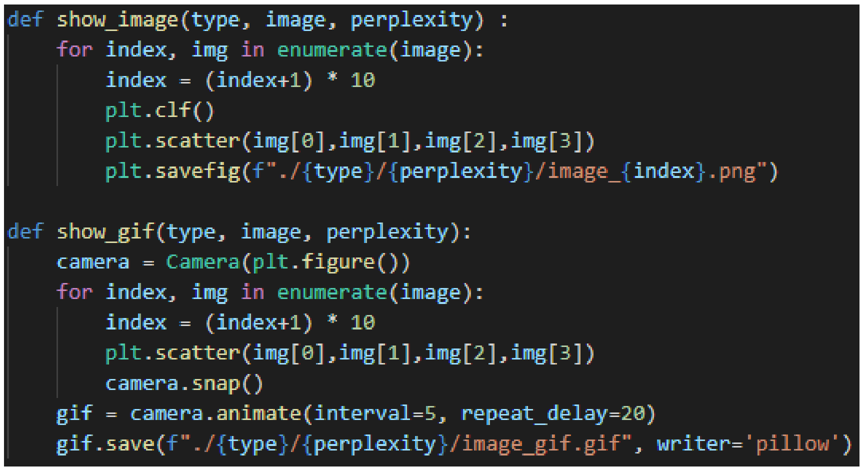
```
if mode == "tsne":
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
else:
    num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))
```



```
if mode =="tsne":
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
else:
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

    - Part 2

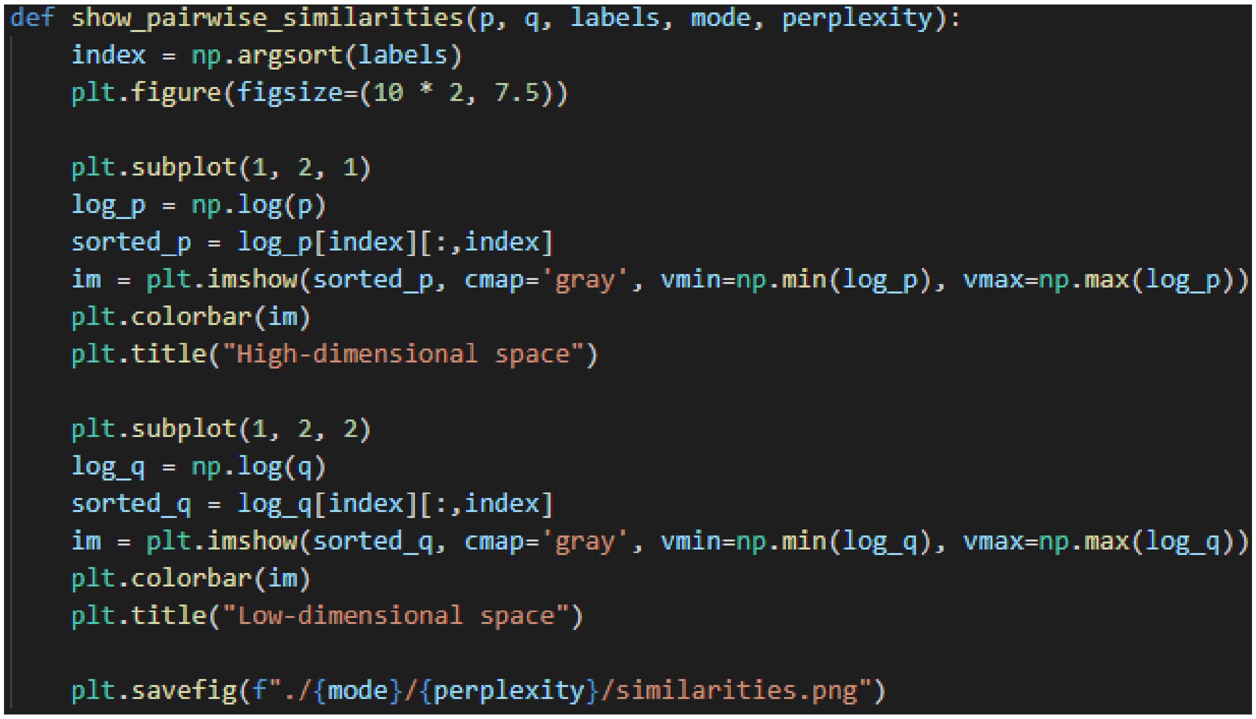
In the sample code, we can get the parameter that the prediction results and its position, we can collect them each 10 epochs, after finfish iterative job. We can plot the visualization of the distribution.



```
def show_image(type, image, perplexity) :
    for index, img in enumerate(image):
        index = (index+1) * 10
        plt.clf()
        plt.scatter(img[0],img[1],img[2],img[3])
        plt.savefig(f"./{type}/{perplexity}/image_{index}.png")

def show_gif(type, image, perplexity):
    camera = Camera(plt.figure())
    for index, img in enumerate(image):
        index = (index+1) * 10
        plt.scatter(img[0],img[1],img[2],img[3])
        camera.snap()
    gif = camera.animate(interval=5, repeat_delay=20)
    gif.save(f"./{type}/{perplexity}/image_gif.gif", writer='pillow')
```

○ Part 3

When the job done, we can get the square matrix P and Q, that represent the probability in the high-dimension and low-dimension. We the also reorder the P and Q by the labels and plot it.



```python
def show_pairwise_similarities(p, q, labels, mode, perplexity):
    index = np.argsort(labels)
    plt.figure(figsize=(10 * 2, 7.5))

    plt.subplot(1, 2, 1)
    log_p = np.log(p)
    sorted_p = log_p[index][:,index]
    im = plt.imshow(sorted_p, cmap='gray', vmin=np.min(log_p), vmax=np.max(log_p))
    plt.colorbar(im)
    plt.title("High-dimensional space")

    plt.subplot(1, 2, 2)
    log_q = np.log(q)
    sorted_q = log_q[index][:,index]
    im = plt.imshow(sorted_q, cmap='gray', vmin=np.min(log_q), vmax=np.max(log_q))
    plt.colorbar(im)
    plt.title("Low-dimensional space")

    plt.savefig(f"./{mode}/{perplexity}/similarities.png")
```

○ Part 4

The part is just tried different parameter in the perplexity.

2. Experiments settings and results & discussion:
   - Kernel Eigenfaces

○ Part 1

This part result is show as the a. part at the above.

○ Part 2

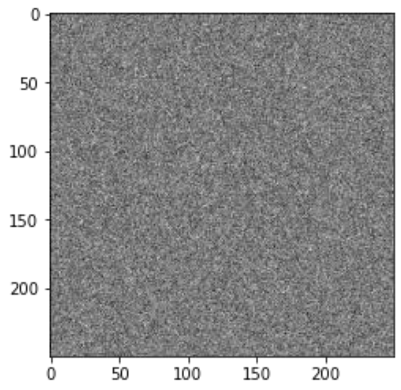
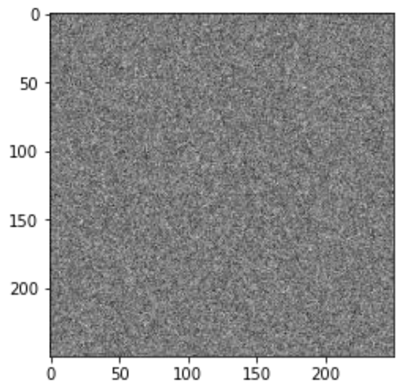
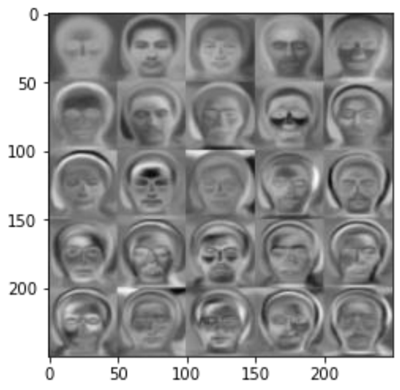
KNN k is setting as 5

	PCA	Kernel PCA			LDA	Kernel LDA		
	simple	linear	poly	RBF	simple	linear	poly	RBF
accuracy	0.8667	0.8	0.8	0.8667	0.9333	0.9	0.8334	0.8

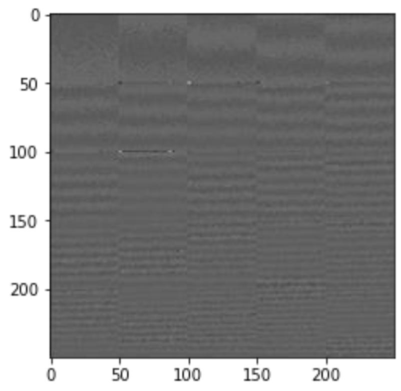
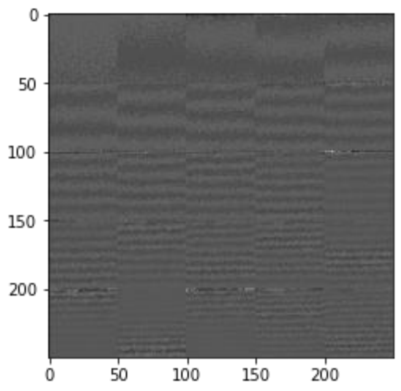
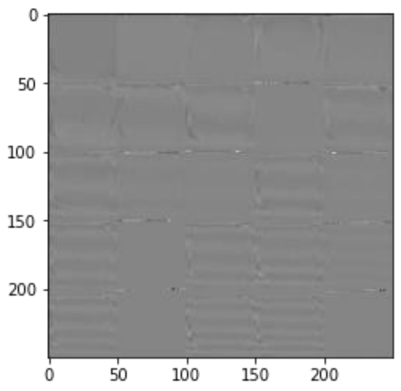
○ Part 3

The kernel PCA and kernel LDA result are show at above, and in the before assignment, when we extend the algorithm to the kernel, we usually can get the higher accuracy and performance, but in the kernel version of PCA and LDA, we didn't see this thing happen, I am curious that or maybe I make something wrong in the implementation.

And others thing is that we can visualize of the eigenface or fisherface, but in the kernel version eigenface and fisherface visualization will seem like the noise, that seem very sensible to me, but in my experiment RBF kernel PCA visualization have some different than other method.

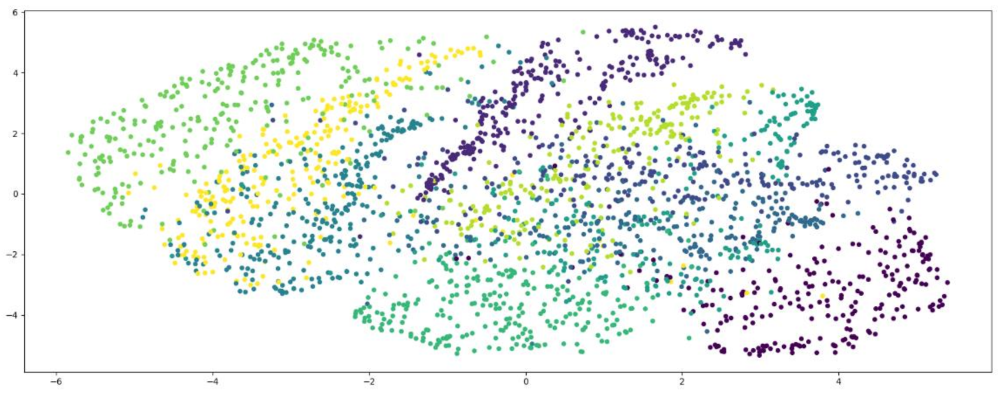
Linear polynomial RBF

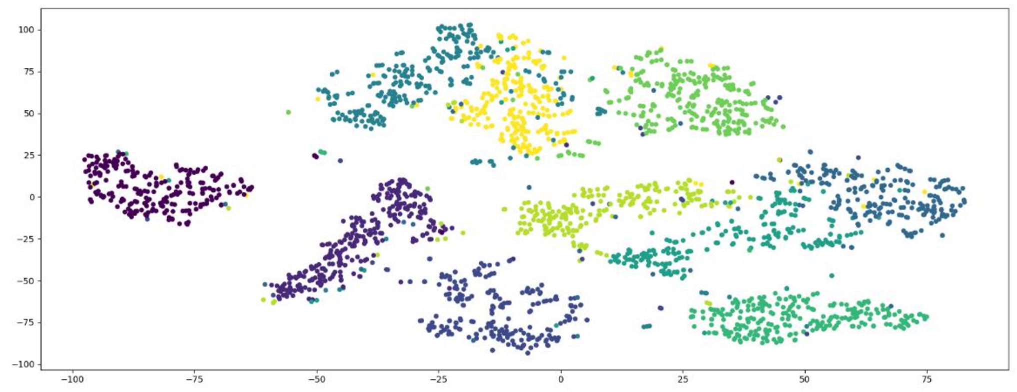




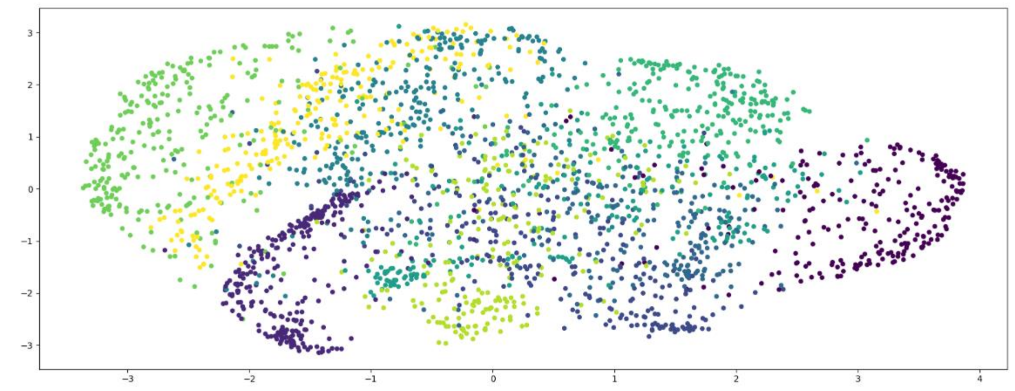
Linear polynomial RBF

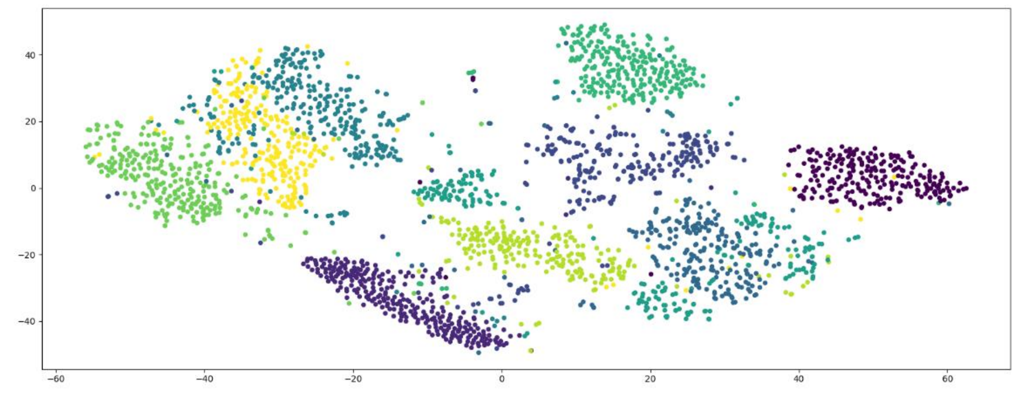
- t-SNE
    - Part 1 & Part 2

The main different between SSNE and t-SNE in the visualization of result, is that SSNE can’t separate well, we can see the first and third image as below, and t-SNE have the better separate performance





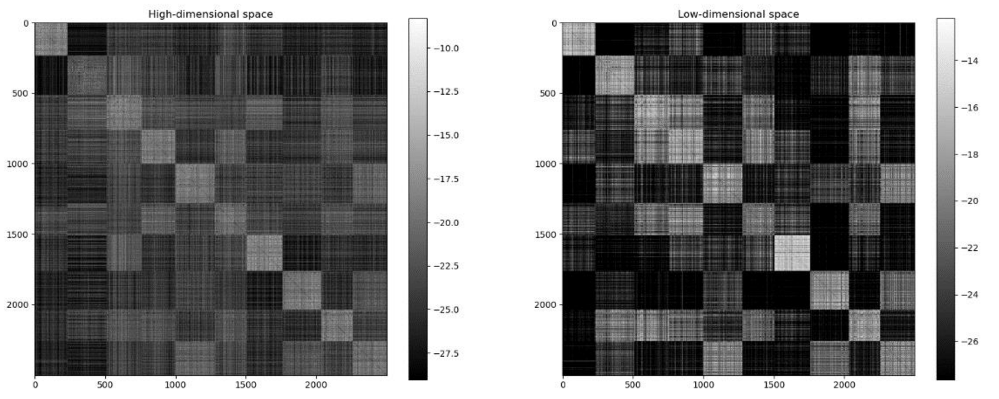




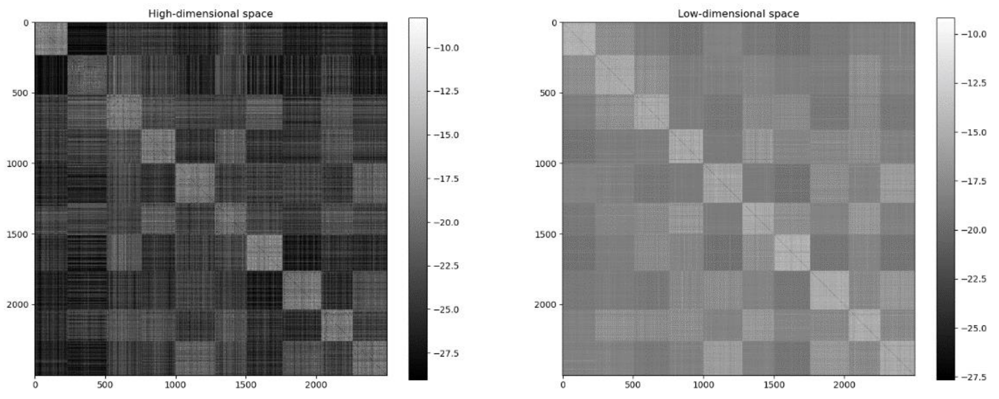
Other experiment and its result and gif will also in the folder in the zip file.

○ Part 3

In the part, we visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space. And these two matrices is reorder version.



Symmetric SNE



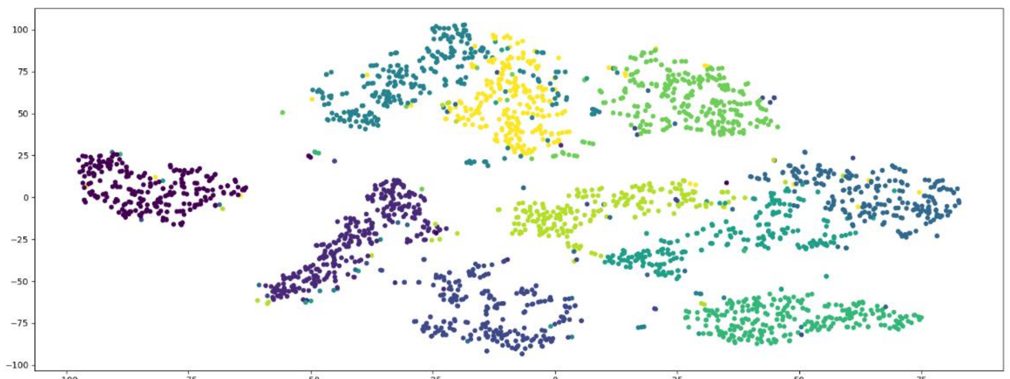
t-SNE

○ Part 4

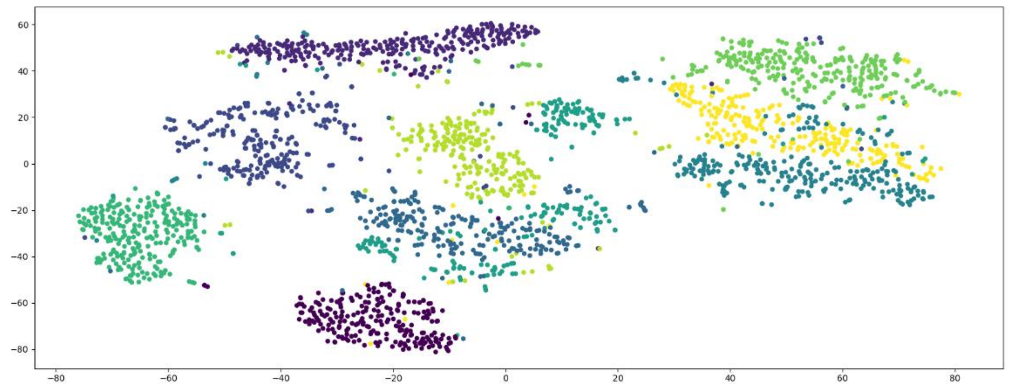
The mainly parameter setting at t-SNE, is perplexity, usually is in the range from 5 to 50, In the fact that the bigger data set usually get more larger perplexity. Perplexity can be seemed like the number of efficient close neighbor point, the larger perplexity we have more closer neighbor point, the sensitive to the small region will be low.

So, we can have following conclusion.

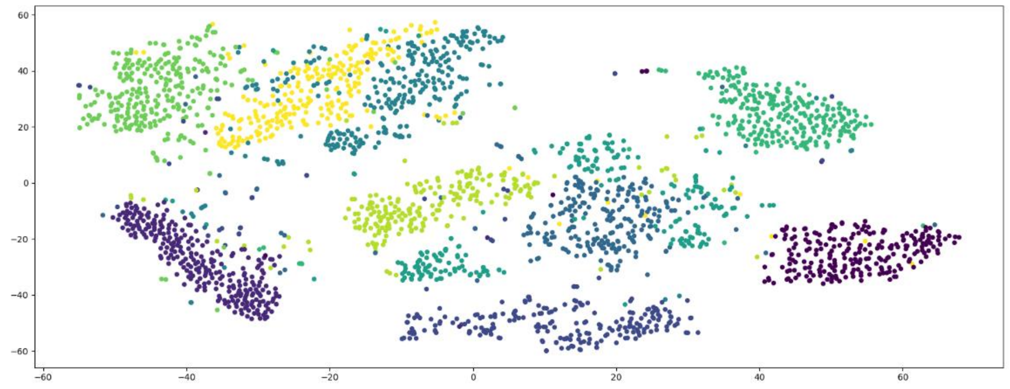
1. Low perplexity: only some few neighbors' point has impact ability, it may divide the same class to different classes.
2. High perplexity: we have clear global structure, but class and class may can't sperate well.



t-SNE with 20 perplexity



t-SNE with 40 perplexity



t-SNE with 60 perplexity

3. Observations and discussion:

In the assignment, the training data set have 135 image face, if I assume that each image is 50 time 50 and reshape it to 2500, we will get, 135 *2500 data matrix, in the kernel mode gram matrix will be 2500 * 2500. But if we increase the number of data set to 500, not change others thing, the gram matrix will still be 2500 * 2500, I am not sure what is the meaning of this gram matrix, and how to explain or just I misunderstand something.

LDA performance is better than PCA, maybe is LDA have the label information, and PCA doesn't have, and PCA want to keep the variance property, but LDA is want to maximum between class and minimum within class, the different opinion LDA maybe more reasonable to this situation. We can found the same result at the kernel version, except the RBF version PCA and LDA.