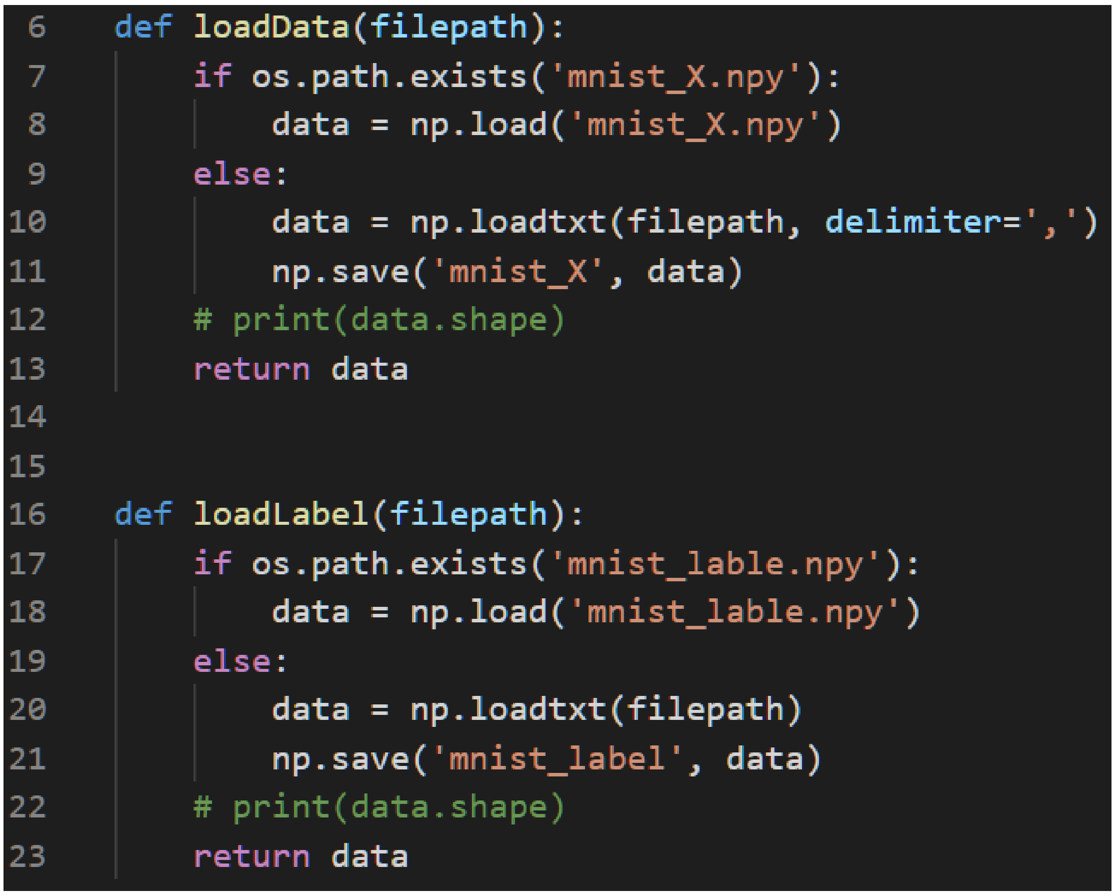
## 1. PCA:

i. Code:

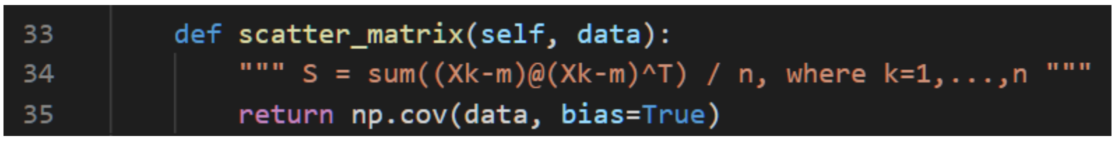
(1) Load data points and labels:



```python
def loadData(filepath):
    if os.path.exists('mnist_X.npy'):
        data = np.load('mnist_X.npy')
    else:
        data = np.loadtxt(filepath, delimiter=',')
        np.save('mnist_X', data)
    # print(data.shape)
    return data


def loadLabel(filepath):
    if os.path.exists('mnist_lable.npy'):
        data = np.load('mnist_lable.npy')
    else:
        data = np.loadtxt(filepath)
        np.save('mnist_label', data)
    # print(data.shape)
    return data
```

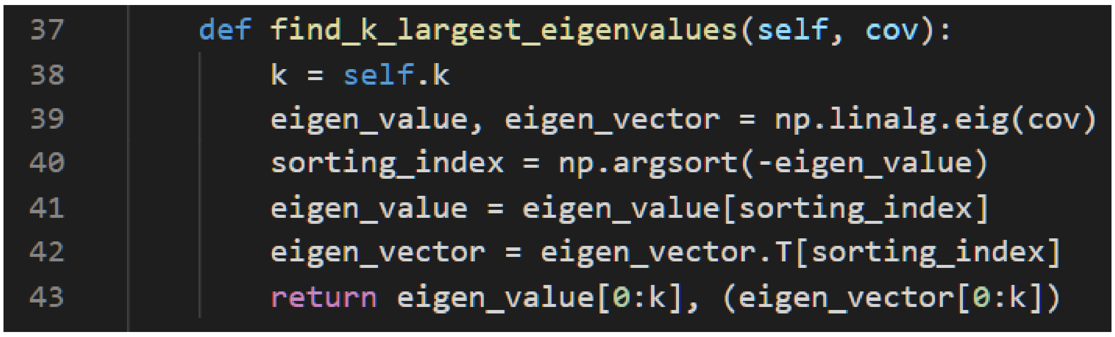
(2) Declare a class **PCA** to implement PCA dimension reduction. The class has following methods:

- Use covariance to calculate scatter matrix.



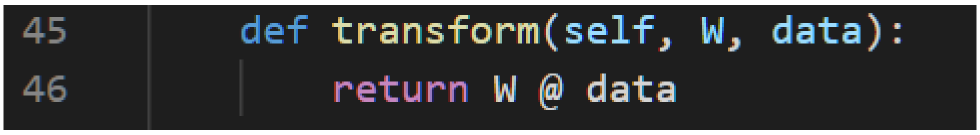
```python
    def scatter_matrix(self, data):
        """ S = sum((Xk-m)@(Xk-m)^T) / n, where k=1,...,n """
        return np.cov(data, bias=True)
```

- Calculate eigenvalues and eigenvectors of scatter matrix and get the first k largest eigenvectors. (These k eigenvectors are principle components.)



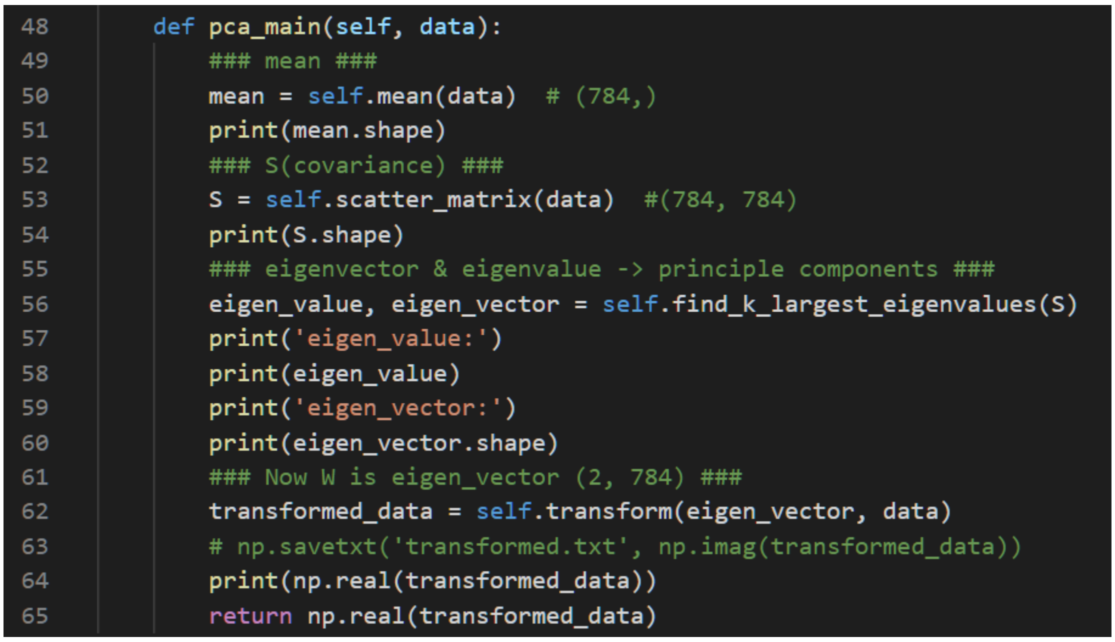
```python
    def find_k_largest_eigenvalues(self, cov):
        k = self.k
        eigen_value, eigen_vector = np.linalg.eig(cov)
        sorting_index = np.argsort(-eigen_value)
        eigen_value = eigen_value[sorting_index]
        eigen_vector = eigen_vector.T[sorting_index]
        return eigen_value[0:k], (eigen_vector[0:k])
```

- Use the eigenvectors to be 2-to-784 dimension W matrix. Transform the samples onto the new subspace: y = $W^TX$.



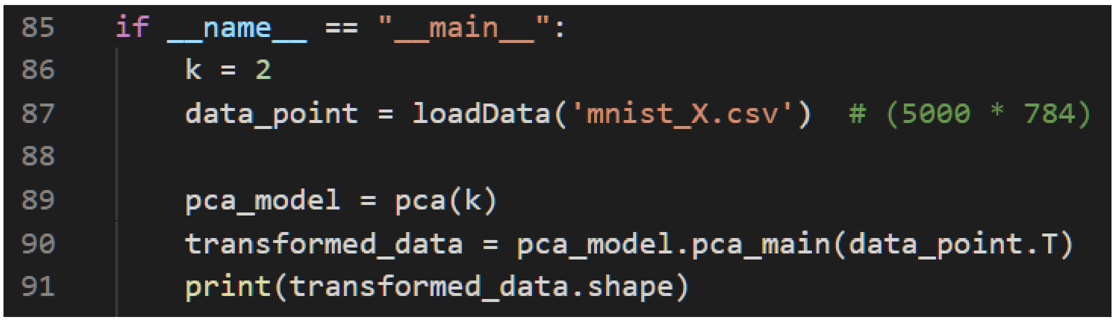
```python
    def transform(self, W, data):
        return W @ data
```

- The main PCA procedure is as following:



```python
    def pca_main(self, data):
        ### mean ###
        mean = self.mean(data)  # (784,)
        print(mean.shape)
        ### S(covariance) ###
        S = self.scatter_matrix(data)  #(784, 784)
        print(S.shape)
        ### eigenvector & eigenvalue -> principle components ###
        eigen_value, eigen_vector = self.find_k_largest_eigenvalues(S)
        print('eigen_value:')
        print(eigen_value)
        print('eigen_vector:')
        print(eigen_vector.shape)
        ### Now W is eigen_vector (2, 784) ###
        transformed_data = self.transform(eigen_vector, data)
        # np.savetxt('transformed.txt', np.imag(transformed_data))
        print(np.real(transformed_data))
        return np.real(transformed_data)
```

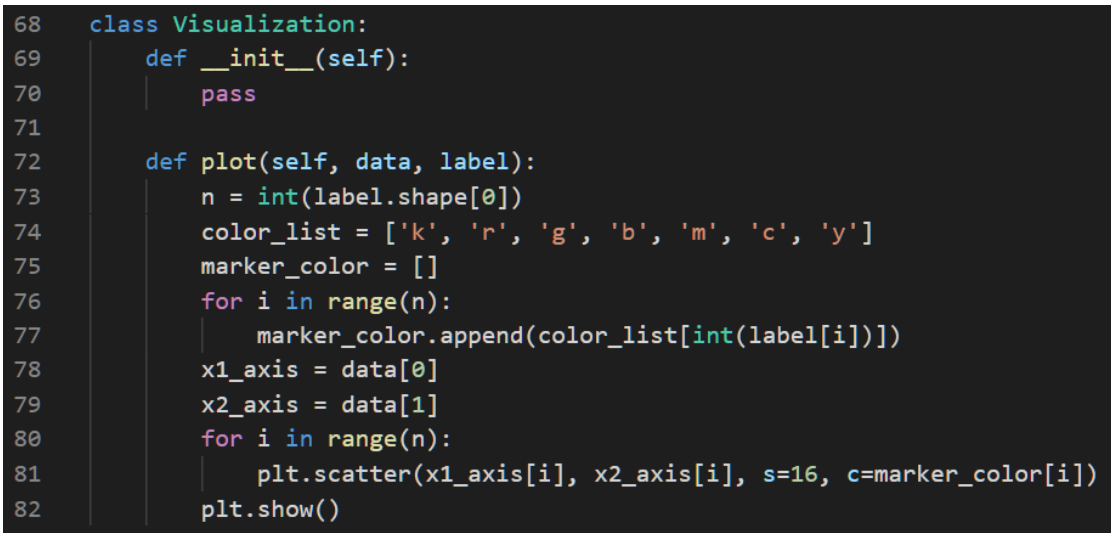
- In main function, use above class as PCA model to implement dimension reduction.



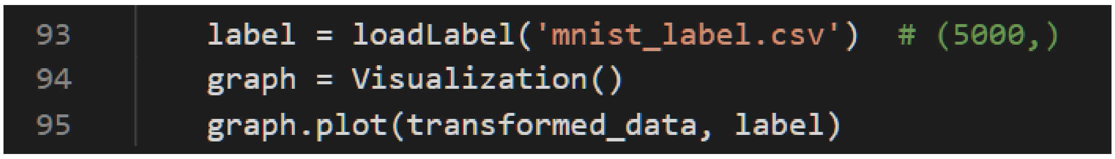
```python
if __name__ == "__main__":
    k = 2
    data_point = loadData('mnist_X.csv')  # (5000 * 784)

    pca_model = pca(k)
    transformed_data = pca_model.pca_main(data_point.T)
    print(transformed_data.shape)
```

(3) Visualization:

- Declare a class **Visualization** to visualize data points on low dimension. For different clusters, give them different colors.

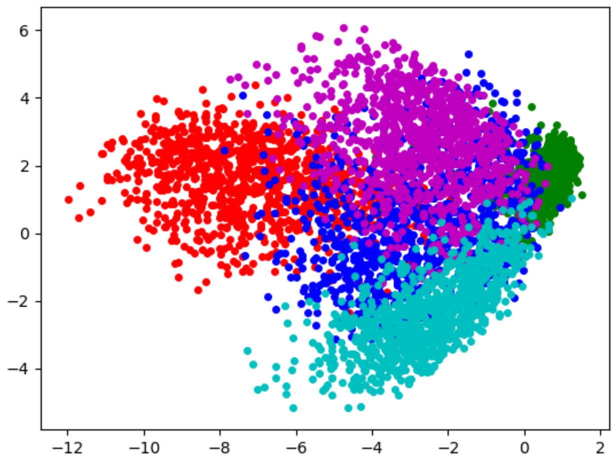


```python
class Visualization:
    def __init__(self):
        pass

    def plot(self, data, label):
        n = int(label.shape[0])
        color_list = ['k', 'r', 'g', 'b', 'm', 'c', 'y']
        marker_color = []
        for i in range(n):
            marker_color.append(color_list[int(label[i])])
        x1_axis = data[0]
        x2_axis = data[1]
        for i in range(n):
            plt.scatter(x1_axis[i], x2_axis[i], s=16, c=marker_color[i])
        plt.show()
```



```python
    label = loadLabel('mnist_label.csv')  # (5000,)
    graph = Visualization()
    graph.plot(transformed_data, label)
```

ii. Results:



iii. Discussion:

The result of PCA indicates those five different clusters in 2-dimension, but each cluster overlaps a lot and cannot be separate easily by linear decision boundary.

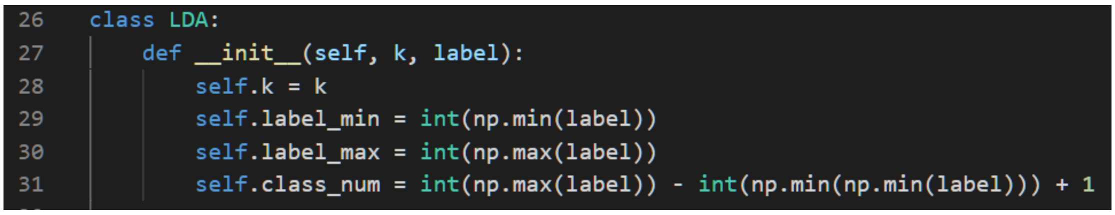
## 2. LDA:

i. Code:

(1) Load data points and labels. (The code is same with PCA's loading data.)

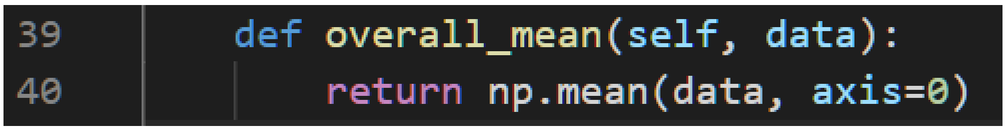
(2) Declare a class **LDA** to implement LDA dimension reduction. The class has following methods and a main method.

- Initial data: initialize class public variables.



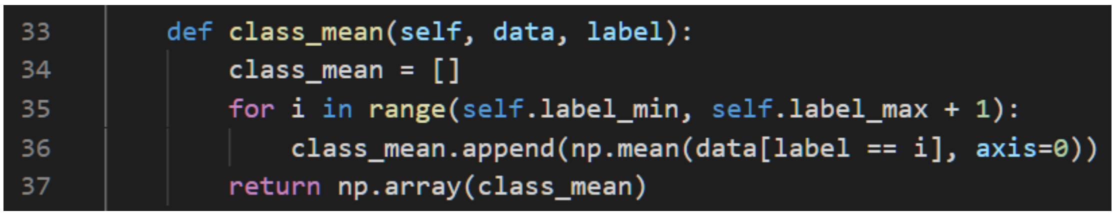
```python
class LDA:
    def __init__(self, k, label):
        self.k = k
        self.label_min = int(np.min(label))
        self.label_max = int(np.max(label))
        self.class_num = int(np.max(label)) - int(np.min(np.min(label))) + 1
```

- Calculate overall mean:



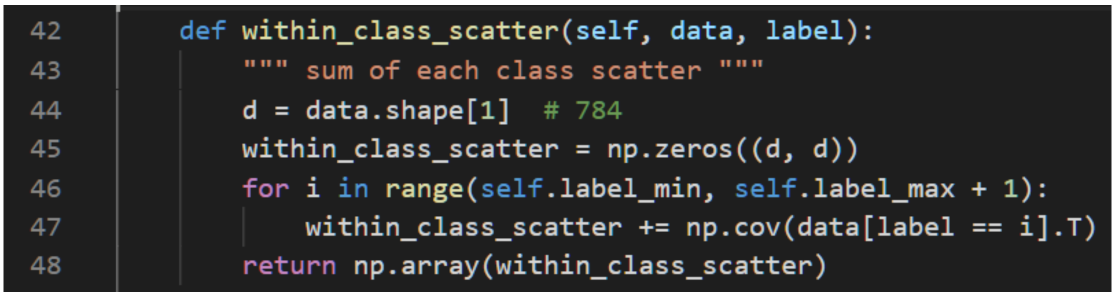
```python
    def overall_mean(self, data):
        return np.mean(data, axis=0)
```

- Calculate class mean:



```python
    def class_mean(self, data, label):
        class_mean = []
        for i in range(self.label_min, self.label_max + 1):
            class_mean.append(np.mean(data[label == i], axis=0))
        return np.array(class_mean)
```

- Calculate within-class scatter matrix Sw:



```python
    def within_class_scatter(self, data, label):
        """ sum of each class scatter """
        d = data.shape[1]  # 784
        within_class_scatter = np.zeros((d, d))
        for i in range(self.label_min, self.label_max + 1):
            within_class_scatter += np.cov(data[label == i].T)
        return np.array(within_class_scatter)
```

- Calculate in-between-class scatter matrix Sb:

```python
    def in_between_class_scatter(self, data, label, class_mean, overall_mean):
        """ sum(nj * (mj-m)@(mj-m)^T), where j=class index """
        class_data_cnt = []
        for i in range(self.label_min, self.label_max + 1):
            class_data_cnt.append(list(label).count(i))
        class_data_cnt = np.array(class_data_cnt)

        d = data.shape[1]  # 784
        in_between_class_scatter = np.zeros((d, d))
        for i in range(self.class_num):
            print(i, ':')
            # print(class_mean[i])
            # print(overall_mean)
            class_mean_col = class_mean[i].reshape(d, 1)
            overall_mean_col = overall_mean.reshape(d, 1)
            tmp = (class_mean_col - overall_mean_col) @ (
                class_mean_col - overall_mean_col).T
            print(tmp.shape)
            print('-------------')
            in_between_class_scatter += class_data_cnt[i] * (
                class_mean_col - overall_mean_col) @ (
                    class_mean_col - overall_mean_col).T
        in_between_class_scatter = np.array(in_between_class_scatter)
        return in_between_class_scatter
```

- Calculate eigenvalues and eigenvectors of **$Sw^{-1}Sb$** and get the first k largest eigenvectors. (These k eigenvectors are principle components.) Note here Sw will become invertible since n<D, so pseudo inverse need to be applied.

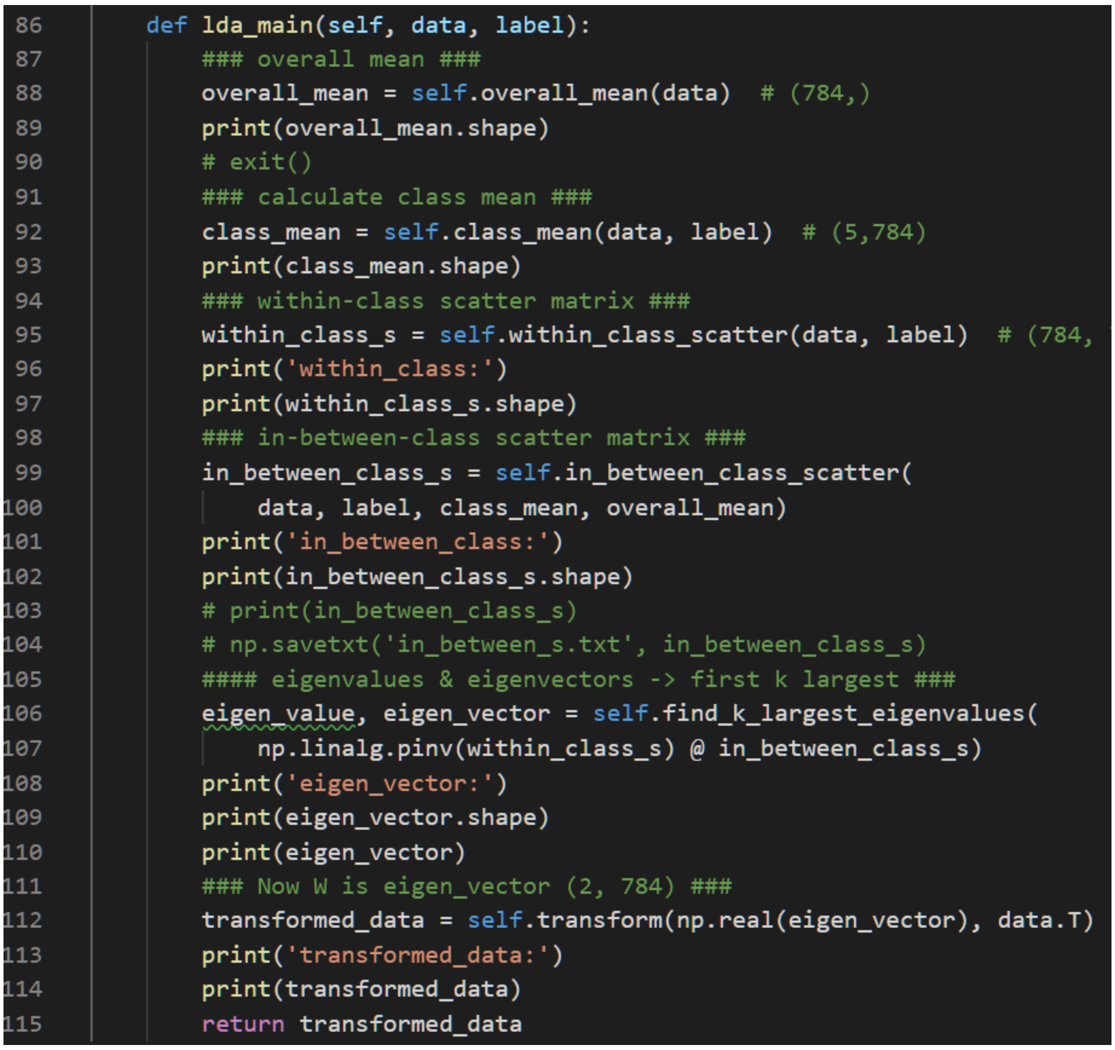
```python
        #### eigenvalues & eigenvectors -> first k largest ###
        eigen_value, eigen_vector = self.find_k_largest_eigenvalues(
            np.linalg.pinv(within_class_s) @ in_between_class_s)
```

```python
    def find_k_largest_eigenvalues(self, cov):
        k = self.k
        eigen_value, eigen_vector = np.linalg.eig(cov)
        sorting_index = np.argsort(-eigen_value)
        eigen_value = eigen_value[sorting_index]
        eigen_vector = eigen_vector.T[sorting_index]
        return eigen_value[0:k], (eigen_vector[0:k])
```

- Use the eigenvectors to be 2-to-784 dimension W matrix. Transform the samples onto the new subspace: $y = W^TX$.

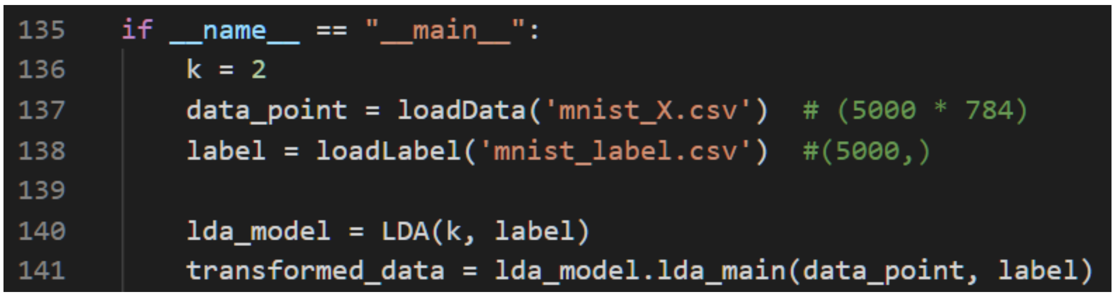
```python
    def transform(self, W, data):
        return W @ data
```

- The main LDA procedure is as following:



```python
    def lda_main(self, data, label):
        ### overall mean ###
        overall_mean = self.overall_mean(data)  # (784,)
        print(overall_mean.shape)
        # exit()
        ### calculate class mean ###
        class_mean = self.class_mean(data, label)  # (5,784)
        print(class_mean.shape)
        ### within-class scatter matrix ###
        within_class_s = self.within_class_scatter(data, label)  # (784,
        print('within_class:')
        print(within_class_s.shape)
        ### in-between-class scatter matrix ###
        in_between_class_s = self.in_between_class_scatter(
            data, label, class_mean, overall_mean)
        print('in_between_class:')
        print(in_between_class_s.shape)
        # print(in_between_class_s)
        # np.savetxt('in_between_s.txt', in_between_class_s)
        #### eigenvalues & eigenvectors -> first k largest ###
        eigen_value, eigen_vector = self.find_k_largest_eigenvalues(
            np.linalg.pinv(within_class_s) @ in_between_class_s)
        print('eigen_vector:')
        print(eigen_vector.shape)
        print(eigen_vector)
        ### Now W is eigen_vector (2, 784) ###
        transformed_data = self.transform(np.real(eigen_vector), data.T)
        print('transformed_data:')
        print(transformed_data)
        return transformed_data
```

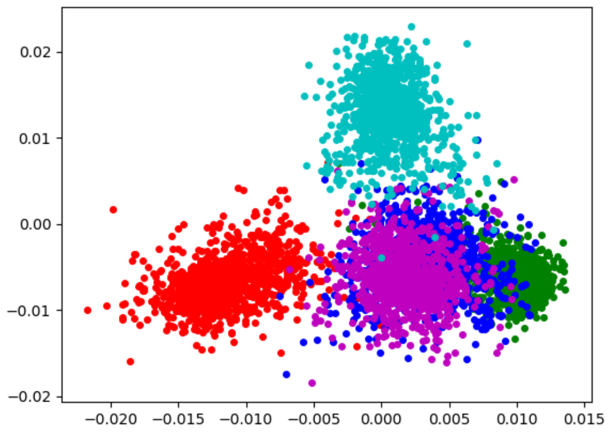
- In main function, use above class as LDA model to implement dimension reduction.



```python
if __name__ == "__main__":
    k = 2
    data_point = loadData('mnist_X.csv')  # (5000 * 784)
    label = loadLabel('mnist_label.csv')  #(5000,)

    lda_model = LDA(k, label)
    transformed_data = lda_model.lda_main(data_point, label)
```

(3) Visualization: the code is the same as PCA's visualization code.

ii. Results:



iii. Discussion:

The main difference between PCA and LDA is that PCA is unsupervised, and LDA is supervised (i.e. PCA doesn't use cluster labels when doing dimension reduction, but LDA does.)

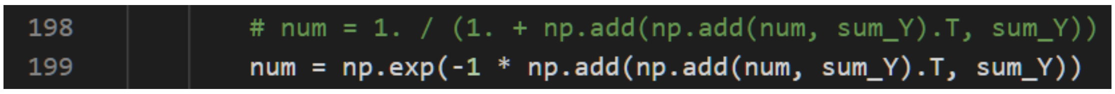
LDA makes data points in same cluster close to each other on low-dimension subspace, and those who are in different cluster are far from each other. In the picture of result, it is easy to find that data points in same cluster are closer than data points in PCA subspace. Also, although there are still some clusters overlapping with each other, but the situation is slighter than that in PCA.

## 3. Symmetric SNE and T-SNE:

i. Code: (iterate 400 times)

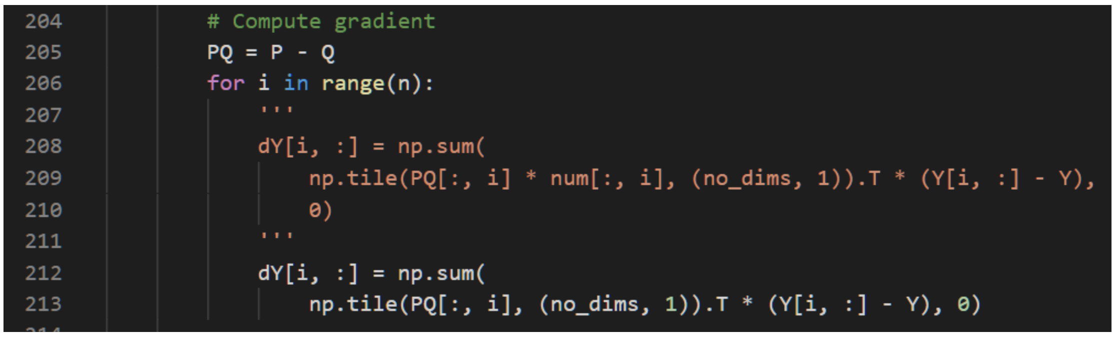
(1) I have changed three things to implement S-SNE:

- Pairwise affinities:



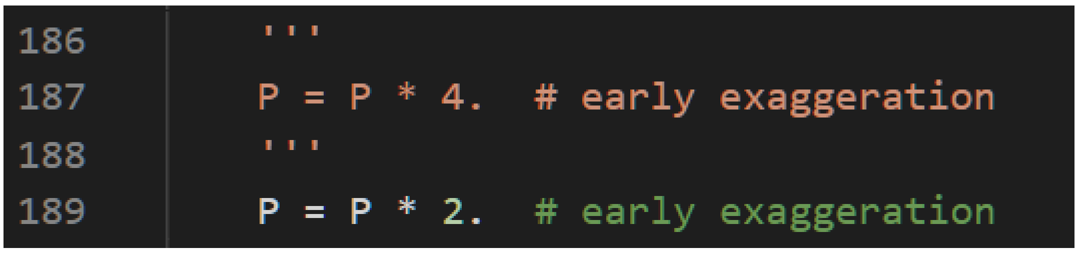
```
        # num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
        num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))
```

- Gradient:



```
        # Compute gradient
        PQ = P - Q
        for i in range(n):
            '''
            dY[i, :] = np.sum(
                np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y),
                0)
            '''
            dY[i, :] = np.sum(
                np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

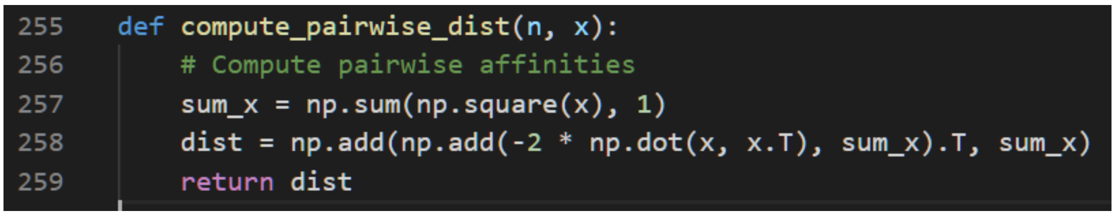
- Early exaggeration:



```python
    '''
    P = P * 4.  # early exaggeration
    '''
    P = P * 2.  # early exaggeration
```

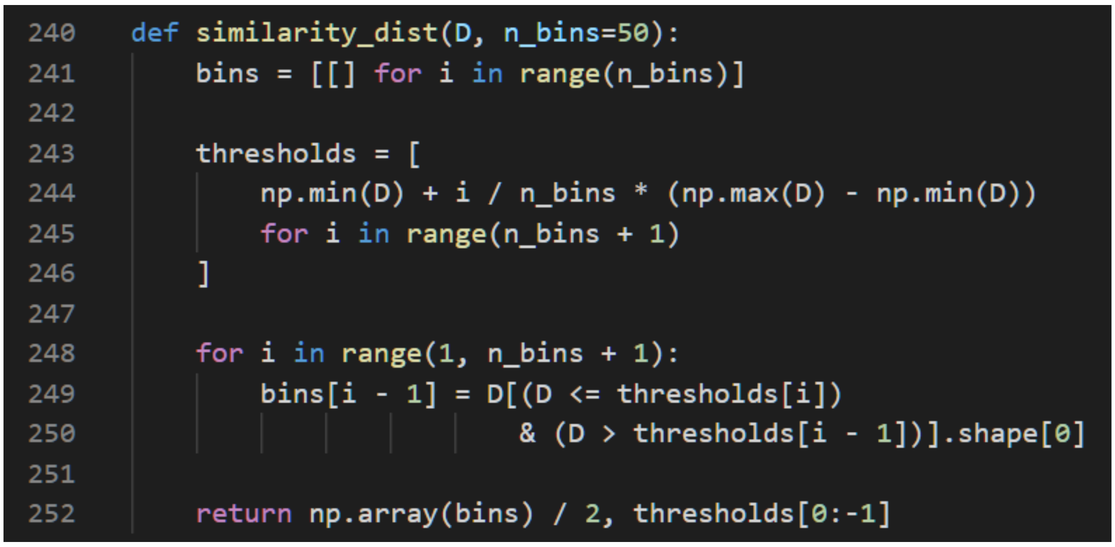
(2) Visualize the distribution of pairwise similarities in both high-dimensional space and low-dimensional space.

- First of all, compute pairwise similarities (distances) using the same equation for calculating Q.



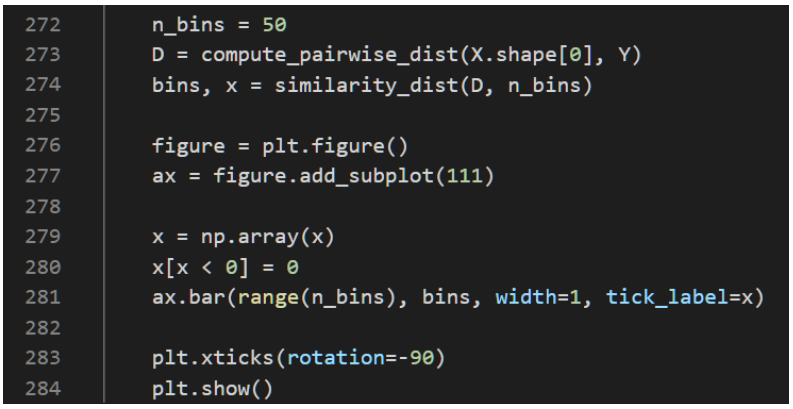
```python
def compute_pairwise_dist(n, x):
    # Compute pairwise affinities
    sum_x = np.sum(np.square(x), 1)
    dist = np.add(np.add(-2 * np.dot(x, x.T), sum_x).T, sum_x)
    return dist
```

- Then use these similarities (distances) to make a histogram. In this histogram, each bin counts data within a certain similarity.



```python
def similarity_dist(D, n_bins=50):
    bins = [[] for i in range(n_bins)]

    thresholds = [
        np.min(D) + i / n_bins * (np.max(D) - np.min(D))
        for i in range(n_bins + 1)
    ]

    for i in range(1, n_bins + 1):
        bins[i - 1] = D[(D <= thresholds[i])
                        & (D > thresholds[i - 1])].shape[0]

    return np.array(bins) / 2, thresholds[0:-1]
```

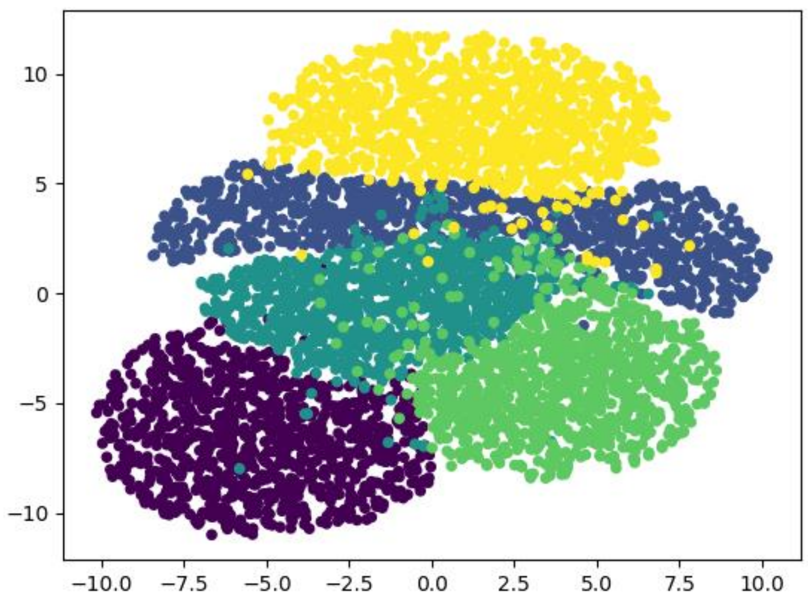
- Finally, visualize this histogram.



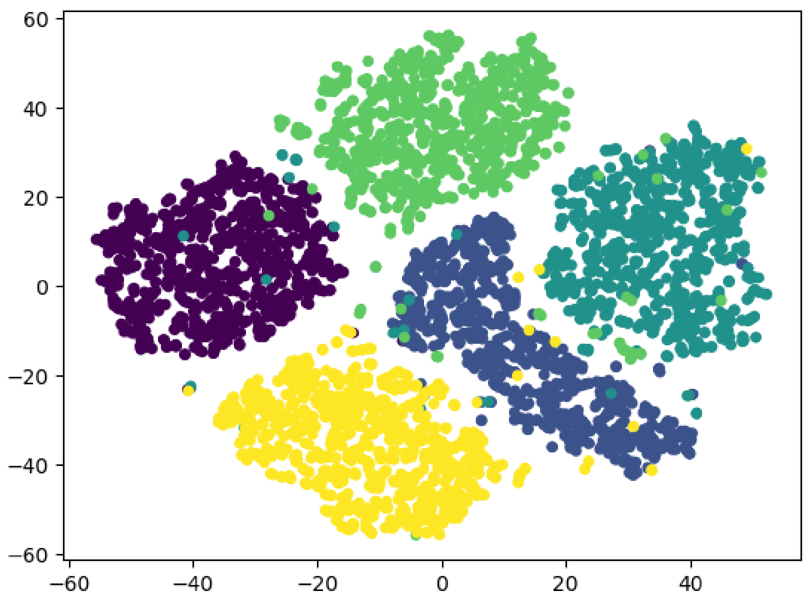
```python
    n_bins = 50
    D = compute_pairwise_dist(X.shape[0], Y)
    bins, x = similarity_dist(D, n_bins)

    figure = plt.figure()
    ax = figure.add_subplot(111)

    x = np.array(x)
    x[x < 0] = 0
    ax.bar(range(n_bins), bins, width=1, tick_label=x)

    plt.xticks(rotation=-90)
    plt.show()
```

ii. Results:

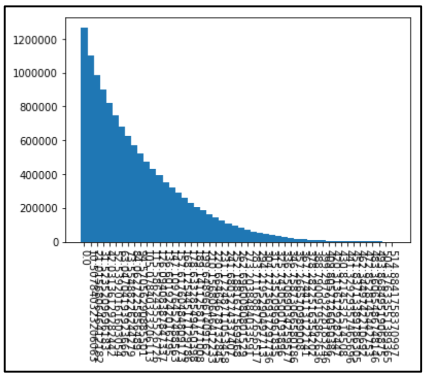
- S-SNE:



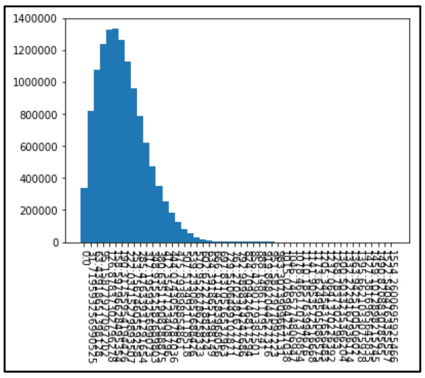
- T-SNE:



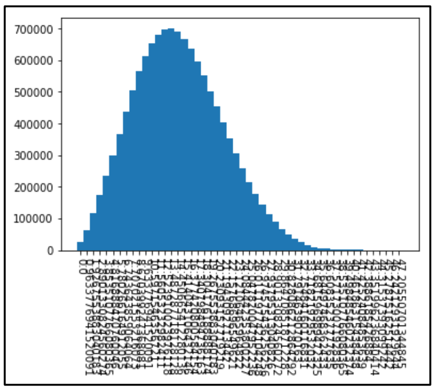
- Pairwise similarities distribution by S-SNE:



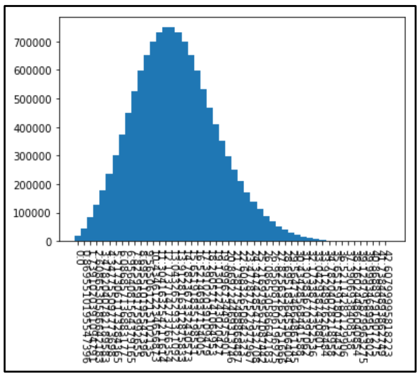
(k=2)



(k=5)

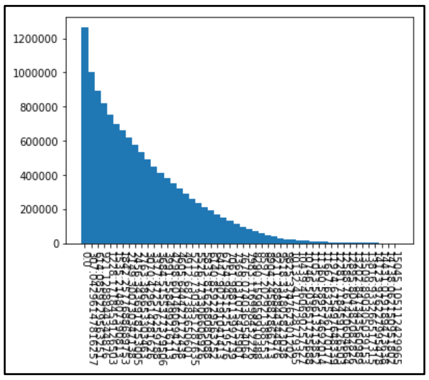


(k=8)

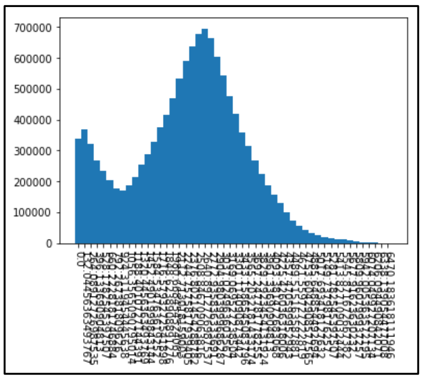


(k=10)

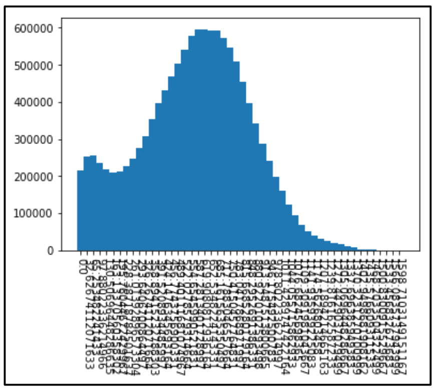
- Pairwise similarities distribution by T-SNE:



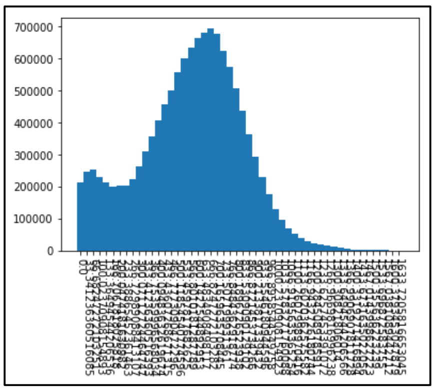
(k=2)



(k=5)



(k=8)



(k=10)

iii. Discussion:

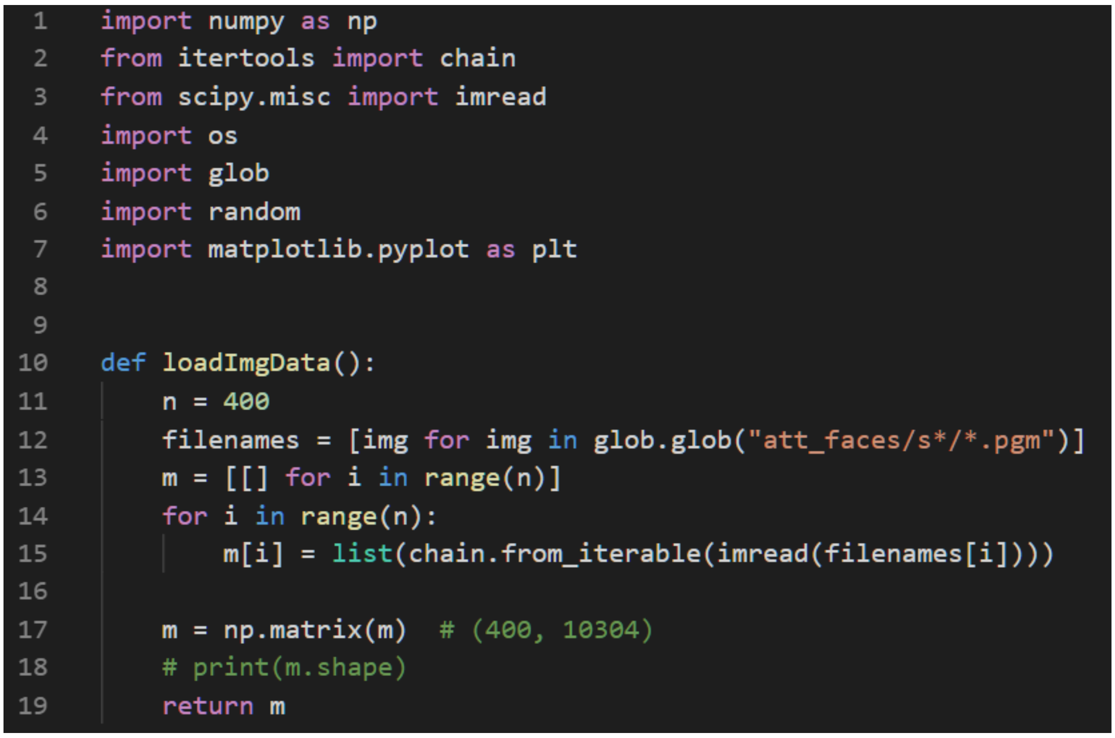
- According to the result figures of T-SNE and S-SNE, overlapping of each cluster data is solved. They preserve the pairwise similarities between hign-D and low-D.

- In S-SNE, the crowded problem can be found easily. In the result figure of S-SNE, the data points are close to each other. According to the figures of pairwise similarities distribution, with k decreasing, most data points concentrate within a certain distance.
- In T-SNE, the crowded problem can be solved.

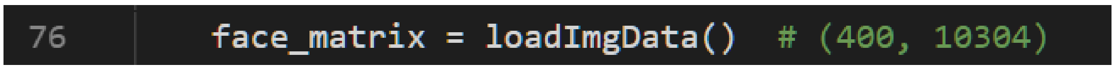
## 4. Eigenface:

i. Code:

(1) Load image data: use **glob** and **imread** (from **scipy.misc**) to read all images in the folders, and then store them in flattened shape into a face matrix.



```
import numpy as np
from itertools import chain
from scipy.misc import imread
import os
import glob
import random
import matplotlib.pyplot as plt


def loadImgData():
    n = 400
    filenames = [img for img in glob.glob("att_faces/s*/*.pgm")]
    m = [[] for i in range(n)]
    for i in range(n):
        m[i] = list(chain.from_iterable(imread(filenames[i])))

    m = np.matrix(m)  # (400, 10304)
    # print(m.shape)
    return m
```



```
    face_matrix = loadImgData()  # (400, 10304)
```

(2) Use the PCA class to implement dimension reduction. Since the dimension d*d(10304 * 10304) is too large for eigenvector calculation and thus the whole training process is very slow, so here I use a speed-up method:

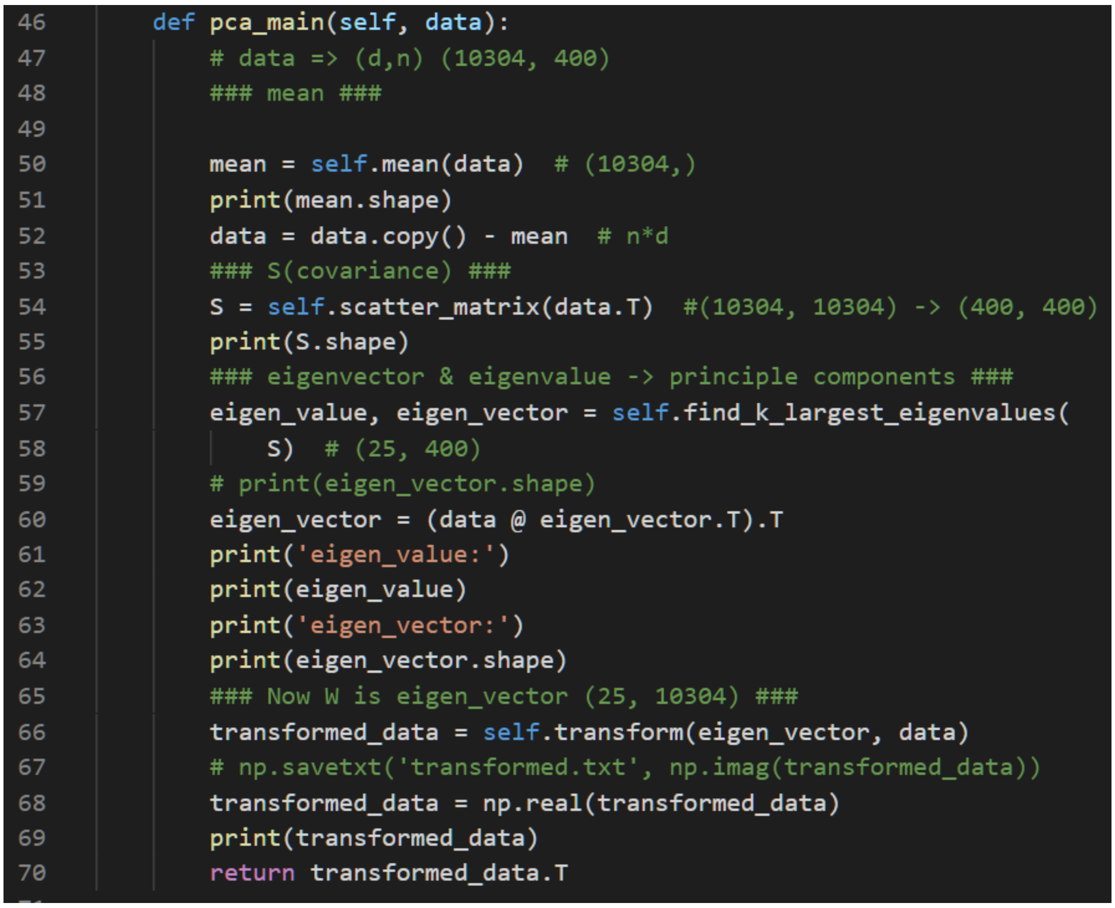
(Reference: this page)

- Calculate difference from data to its mean: data = data – mean.
- Calculate new scatter matrix: C = cov(data) -> C’ = cov(data.T)
- Use C’ to calculate k’s largest eigenvectors: Ei.

- Let original eigenvectors (k's largest eigenvectors from C) be Vi.
  Vi = (data @ Ei).T
  Proof is shown below:

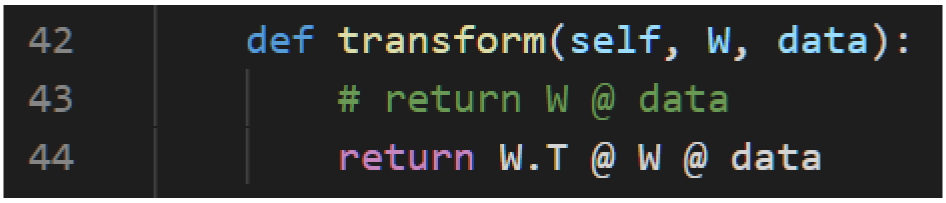
$$
\begin{array}{rrcl}
 & C' \cdot e_i = \lambda \cdot e_i & | & C' = \Phi^T \cdot \Phi \\
\Leftrightarrow & \Phi^T \cdot \Phi \cdot e_i = \lambda \cdot e_i & | & \text{multiply from the left with } \Phi \\
\Leftrightarrow & \Phi \cdot \Phi^T \cdot \Phi \cdot e_i = \lambda \cdot \Phi \cdot e_i & | & C = \Phi \cdot \Phi^T \\
\Leftrightarrow & C \cdot \Phi \cdot e_i = \lambda \cdot \Phi \cdot e_i & | & \text{define } v_i = \Phi \cdot e_i \\
\Leftrightarrow & C \cdot v_i = \lambda \cdot v_i & & 
\end{array}
$$

http://blog.csdn.net/zouxy09



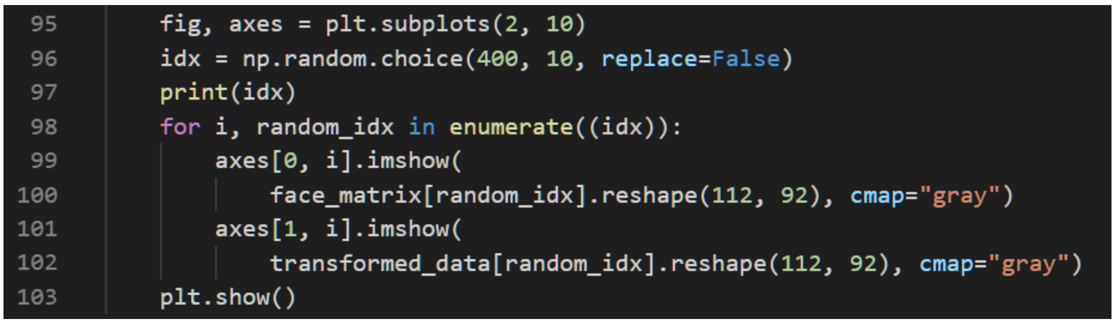
```python
    def pca_main(self, data):
        # data => (d,n) (10304, 400)
        ### mean ###

        mean = self.mean(data)  # (10304,)
        print(mean.shape)
        data = data.copy() - mean  # n*d
        ### S(covariance) ###
        S = self.scatter_matrix(data.T)  #(10304, 10304) -> (400, 400)
        print(S.shape)
        ### eigenvector & eigenvalue -> principle components ###
        eigen_value, eigen_vector = self.find_k_largest_eigenvalues(
            S)  # (25, 400)
        # print(eigen_vector.shape)
        eigen_vector = (data @ eigen_vector.T).T
        print('eigen_value:')
        print(eigen_value)
        print('eigen_vector:')
        print(eigen_vector.shape)
        ### Now W is eigen_vector (25, 10304) ###
        transformed_data = self.transform(eigen_vector, data)
        # np.savetxt('transformed.txt', np.imag(transformed_data))
        transformed_data = np.real(transformed_data)
        print(transformed_data)
        return transformed_data.T
```

(3) Transform data into new subspace: $y = W^TWx$



```python
    def transform(self, W, data):
        # return W @ data
        return W.T @ W @ data
```

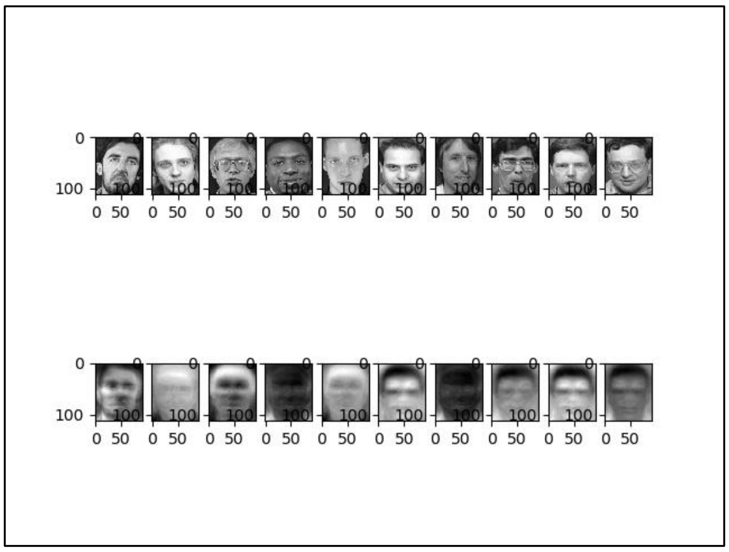
(4) Visualization: Randomly choose 10 images. Rescale the transformed data into image shape (112*92) and show it with the original image.

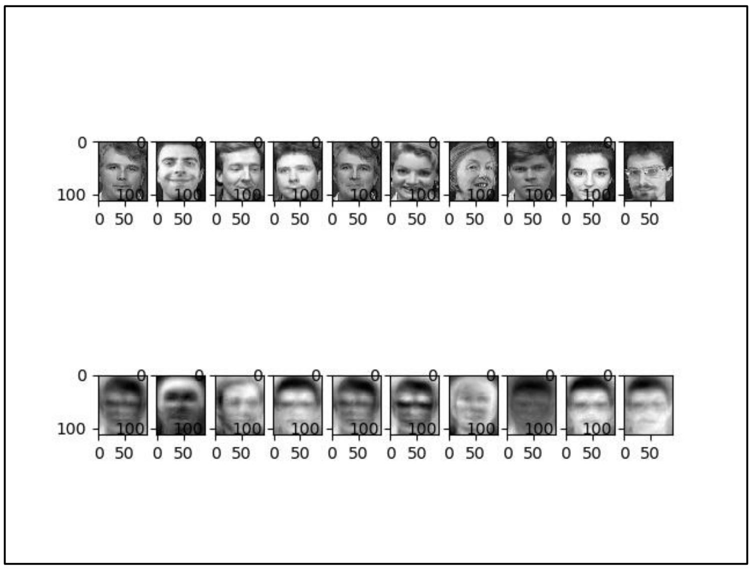


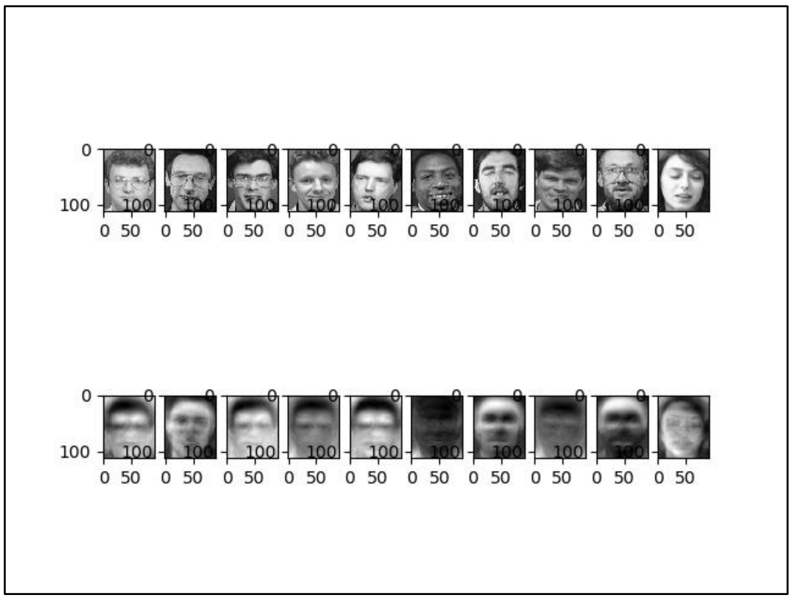
```python
    fig, axes = plt.subplots(2, 10)
    idx = np.random.choice(400, 10, replace=False)
    print(idx)
    for i, random_idx in enumerate((idx)):
        axes[0, i].imshow(
            face_matrix[random_idx].reshape(112, 92), cmap="gray")
        axes[1, i].imshow(
            transformed_data[random_idx].reshape(112, 92), cmap="gray")
    plt.show()
```

ii. Results:

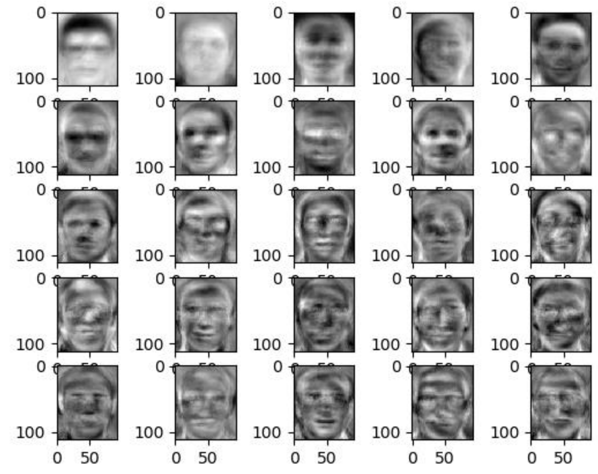
(1) Randomly choose 10 images. The results of different trials are as following:







(2) The first 25 eigenfaces:



iii. Discussion:

When calculating eigenfaces, without the speed-up trick, the training time is really long (more than 10 minutes). After speeding-up, the pca model can thus calculating high-dimension data quickly, which makes it possible to deal with a big amount of faces.

Eigenfaces contains some “messages” of a certain person, which is useful for face recognition. The result of those 10 randomly chosen faces are mostly well-recognized.