Machine Learning Homework 7 Technical Report

tags: `Course1082` `IOC5191`

網工所 0856523 許尹睿

Since there are a lot of figures and GIFs to be showed, I place all of them in directory Result instead of directly putting all of them in the report.

Naming Principles for each figure and GIF are explained in **Appendix**, which is for TA's convenience.

Kernel Eigenfaces/Fisherfaces

PCA & Kernel PCA

Extract data

```
# read the input images
def read_images(path):
    dataset, label = list(), list()
    for data in os.listdir(path):
        with Image.open(os.path.join(path, data)) as image:
            dataset.append(np.array(image).flatten())
            label.append(int(re.findall("\d+", data)[0]))

    return np.array(dataset), np.array(label)

train_path = os.path.join('Yale_Face_Database', 'Training')
test_path = os.path.join('Yale_Face_Database', 'Testing')

train_images, train_label = read_images(train_path)
train_images = train_images[np.argsort(train_label), :]
train_label = np.sort(train_label)
test_images, test_label = read_images(test_path)
test_images = test_images[np.argsort(test_label), :]
test_label = np.sort(test_label)
```

read_image function transforms the training and testing images into the 2-D numpy array, and stores their label into the other 1-D numpy array.
The shape of train_images and test_images is (135, 45045) and (30, 45045). The shape of train_labels and test_labels is (135,) and (30,)

Implement PCA

```
def PCA(images, no_dims=25):
    (n, d) = images.shape
    images = images - np.tile(np.mean(images, 0), (n, 1))
    eigen_values, eigen_vectors = np.linalg.eigh(np.dot(images, images.T))
    print("Computing Eigenvalues...")
    principle_component = eigen_vectors[:, np.argsort(eigen_values)[::-1][0:
no_dims]].real.astype(np.float32)
    principle_component = np.dot(images.T,
principle_component).real.astype(np.float32)
    norm = np.linalg.norm(principle_component, axis=0)
    principle_component = np.divide(principle_component, norm)

    return principle_component

principle_component = PCA(train_images)
```

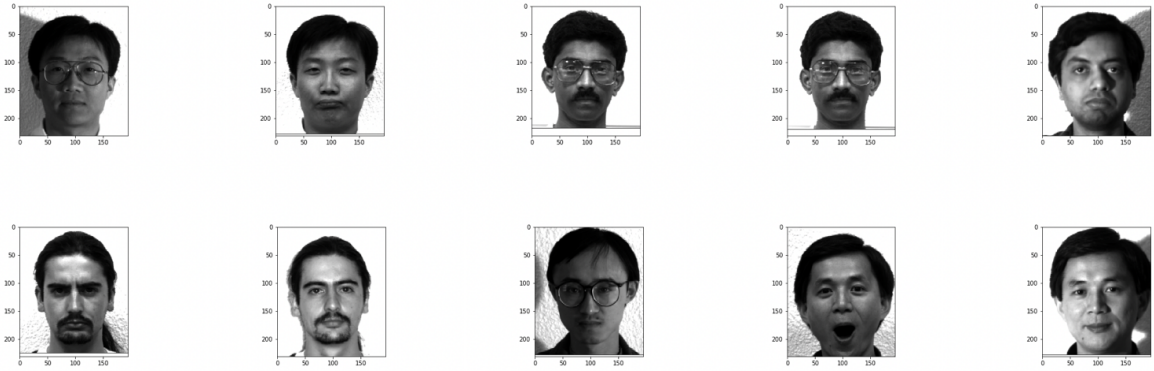
PCA function extracts the principle components from the images, which are the eigenvectors corresponding to the largest 25 eigenvalues.
Because images.T@images will be a very large matrix, we find the eigenvalues and eigenvectors of images@images.T. These corresponding eigenvectors then be multiplied by images again to represent the partial eigenvectors of images.T@images.

Reconstruct the images

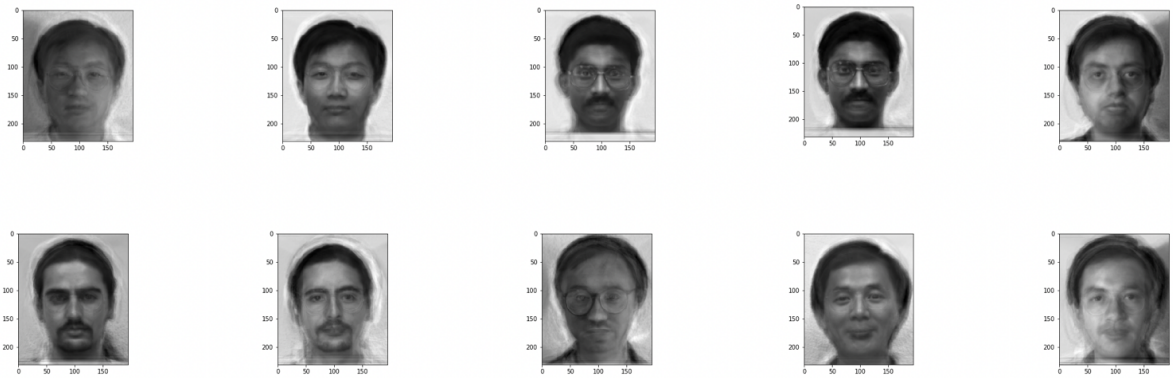
```
result_path = os.path.join('Result', 'PCA')
def plot_reconstruct(reconstruct, origin, index):
    for idx in index:
        plt.clf()
        plt.imshow(reconstruct[idx].reshape((231, 195)), plt.cm.gray)
        plt.savefig(f"{result_path}/reconstruct_{idx}.png")

        plt.clf()
        plt.imshow(origin[idx].reshape((231, 195)), plt.cm.gray)
        plt.savefig(f"{result_path}/origin_{idx}.png")

mean = np.mean(train_images, 0)
high_images = np.dot(train_images - mean, np.dot(principle_component,
principle_component.T)) + mean

index = np.random.randint(len(train_images), size=10)
plot_reconstruct(high_images, train_images, index)
```

plot_reconstruct function randomly pick 10 images to show their reconstruction.

- The original images:



- The images after reconstruction:



Plot Eigenfaces

```
result_path = os.path.join('Result', 'PCA')
def plot_eigenfaces(principle_component):
    for idx in range(len(principle_component)):
        plt.clf()
        plt.imshow(principle_component[idx].reshape((231, 195)), plt.cm.gray)
        plt.savefig(f"{result_path}/eigenfaces_{idx}.png")

plot_eigenfaces(principle_component.T)
```

plot_eigenfaces function shows the first 25 eigenfaces. The following figure shows the result:

Compute the Performance

```
def predict(train_data, test_data, k, label):
    prd_result = []
    for data in test_data:
        distance = np.linalg.norm(train_data - data, axis=1)
        idx = np.argsort(distance)
        neighbors = label[idx][:k]
        prd_result.append(np.bincount(neighbors).argmax())

    return np.array(prd_result)

train_low_images = np.dot(train_images - mean, principle_component)
test_low_images = np.dot(test_images - mean, principle_component)
predict_result = predict(train_low_images, train_low_images, 5, train_label)
print(f'Predict result of Training data set is
{len(predict_result[predict_result == train_label]) / len(train_label)}')
predict_result = predict(train_low_images, test_low_images, 5, train_label)
print(f'Predict result of Testing data set is
{len(predict_result[predict_result == test_label]) / len(test_label)}')
```

predict function uses k nearest neighbor to classify which subject the training and testing image belongs to. The result is listed in the following table:

	Training	Testing
Accuracy	0.8740740740740741	0.9

Implement Kernel PCA

```
def make_kernel(images, kernel):
    if kernel == 'rbf':
        similarity = squareform(pdist(images), 'sqeuclidean')
        gram = np.exp(-gamma * similarity)
    elif kernel == 'linear':
        gram = np.dot(images, images.T)
    return gram

def kernel_PCA(images, no_dims=25):
    kernels = ['linear', 'rbf']
    for kernel in kernels:
        K = make_kernel(images, kernel)
        N = K.shape[0]
        one = np.ones((N, N)) / N
        Kc = K - np.dot(one, K) - np.dot(K, one) + np.dot(np.dot(one, K), one)
        eigen_values, eigen_vectors = np.linalg.eigh(Kc)
        principle_component = eigen_vectors[:, np.argsort(eigen_values)[::-1]
[0: no_dims]].real.astype(np.float32)

        low_images = np.dot(K, principle_component)

        train_low_images = low_images[:len(train_images), ]
        test_low_images = low_images[len(train_images):, ]

        predict_result = predict(train_low_images, train_low_images, 5,
train_label)
        print(
            f'Predict result of Training data set using {kernel} kernel is
{len(predict_result[predict_result == train_label]) / len(train_label)}')
        predict_result = predict(train_low_images, test_low_images, 5,
train_label)
        print(
            f'Predict result of Testing data set using {kernel} kernel is
{len(predict_result[predict_result == test_label]) / len(test_label)}')

images = np.concatenate((train_images, test_images), axis=0)
kernel_PCA(images)
```

make_kernel function implements two types of kernel, which are RBF kernel and linear kernel. **kernel_PCA** function implements the kernel PCA algorithm and extracts the principle components from the images, then using this principle components to do the dimensionality resuction and compute the performance. The accuracy is listed in the following table:

	Training	Testing
linear	0.3111111111111111	0.1
rbf	0.8148148148148148	0.8333333333333334

LDA & Kernel LDA

Extract data

```
# read the input images
def read_images(path):
    dataset, label = list(), list()
    for data in os.listdir(path):
        with Image.open(os.path.join(path, data)).resize((60, 60),
Image.ANTIALIAS) as image:
            dataset.append(np.array(image).flatten())
            label.append(int(re.findall("\d+", data)[0]))

    return np.array(dataset), np.array(label)

train_path = os.path.join('Yale_Face_Database', 'Training')
test_path = os.path.join('Yale_Face_Database', 'Testing')

train_images, train_label = read_images(train_path)
train_images = train_images[np.argsort(train_label), :]
train_label = np.sort(train_label)
test_images, test_label = read_images(test_path)
test_images = test_images[np.argsort(test_label), :]
test_label = np.sort(test_label)
```

read_image function transforms and compresses the training and testing images into the 2-D numpy array, then stores their label into the other 1-D numpy array.
The shape of train_images and test_images is (135, 3600) and (30, 3600). The shape of train_labels and test_labels is (135,) and (30,)

Implement LDA

```
def LDA(images, no_dims=25):
    print('Computing Meam...')
    all_mean = np.mean(images, axis=0).reshape(-1, 1).T
    class_mean = []
    for idx in range(15):
        rows = images[9 * idx:9 * (idx + 1), :]
        class_mean.append(np.mean(rows, axis=0))
    class_mean = np.array(class_mean).astype(np.float32)

    print('Computing SW...')
```

```
    (n, d) = images.shape
    temp = images.copy().astype(np.float32)
    for data in range(n):
        temp[data, :] -= class_mean[data // 9, :]
    within_class = np.dot(temp.T, temp)

    print('Computing SB...')
    (n, d) = class_mean.shape
    temp = class_mean.copy()
    for data in range(n):
        temp[data, :] -= all_mean[0, :]
    between_class = np.dot(temp.T, temp)
    between_class *= 9

    print('Computing Eigenvector...')
    eigen_values, eigen_vectors =
np.linalg.eigh(np.dot(np.linalg.pinv(within_class), between_class))
    print('Computing Principle Component...')
    principle_component = eigen_vectors[:, np.argsort(eigen_values)[::-1][0:
no_dims]].real.astype(np.float32)

    return principle_component

principle_component = LDA(train_images)
```

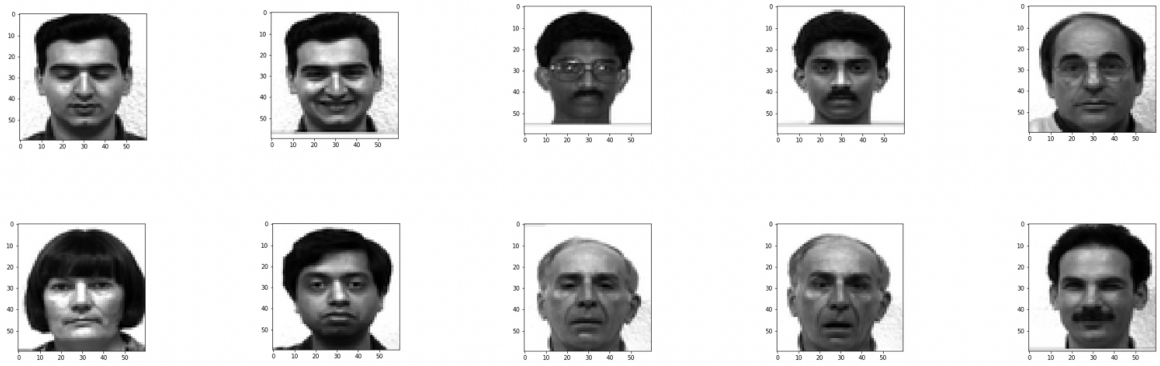
LDA function calculates the between-calss scatter and within-class scatter from the images and uses them to extract the principle components, which are the eigenvectors corresponding to the largest 25 eigenvalues.

Reconstruct the images

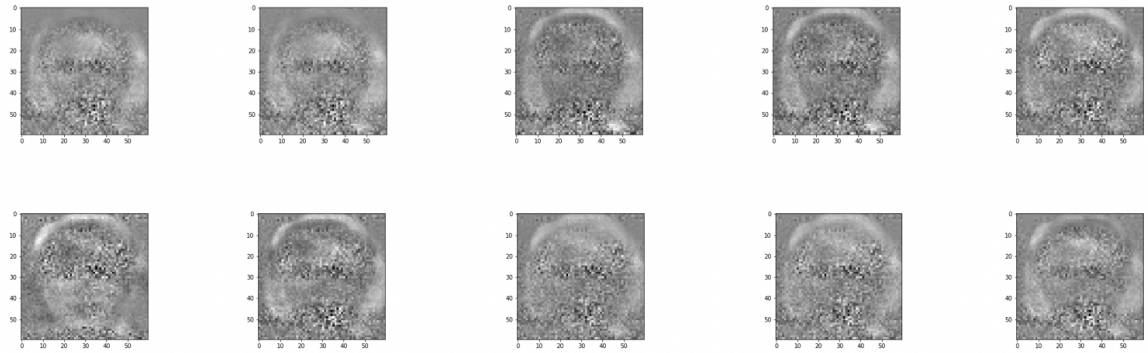
```
result_path = os.path.join('Result', 'LDA')
def plot_reconstruct(reconstruct, origin, index):
    for idx in index:
        plt.clf()
        plt.imshow(reconstruct[idx].reshape((60, 60)), plt.cm.gray)
        plt.savefig(f"{result_path}/reconstruct_{idx}.png")

        plt.clf()
        plt.imshow(origin[idx].reshape((60, 60)), plt.cm.gray)
        plt.savefig(f"{result_path}/origin_{idx}.png")

low_images = np.dot(train_images, principle_component)
high_images = np.dot(low_images, principle_component.T)

index = np.random.randint(len(train_images), size=10)
plot_reconstruct(high_images, train_images, index)
```

plot_reconstruct function randomly pick 10 images to show their reconstruction, but the images after reconstruction are not obvious.

- The original images:



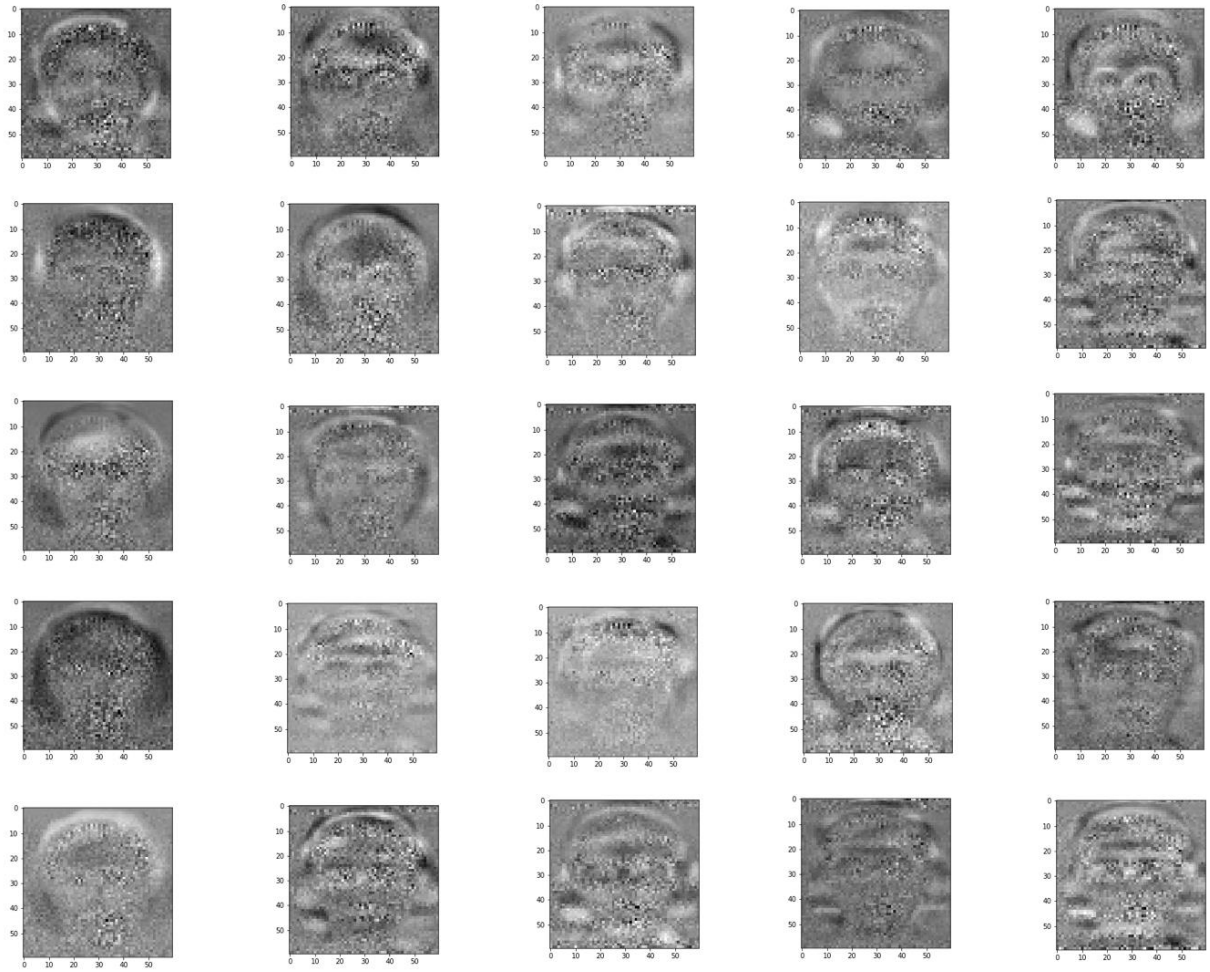
- The images after reconstruction:



Plot Fisherfaces

```
result_path = os.path.join('Result', 'LDA')
def plot_fisherfaces(principle_component):
    for idx in range(len(principle_component)):
        plt.clf()
        plt.imshow(principle_component[idx].reshape((60, 60)), plt.cm.gray)
        plt.savefig(f"{result_path}/fisherfaces_{idx}.png")

plot_fisherfaces(principle_component.T)
```

plot_fisherfaces function shows the first 25 fisherfaces. The following figure shows the result:



Compute the Performance

```
def predict(train_data, test_data, k, label):
    prd_result = []
    for data in test_data:
        distance = np.linalg.norm(train_data - data, axis=1)
        idx = np.argsort(distance)
        neighbors = label[idx][:k]
        prd_result.append(np.bincount(neighbors).argmax())

    return np.array(prd_result)

train_low_images = np.dot(train_images, principle_component)
test_low_images = np.dot(test_images, principle_component)
predict_result = predict(train_low_images, train_low_images, 5, train_label)
print(f'Predict result of Training data set is
{len(predict_result[predict_result == train_label]) / len(train_label)}')
predict_result = predict(train_low_images, test_low_images, 5, train_label)
print(f'Predict result of Testing data set is
{len(predict_result[predict_result == test_label]) / len(test_label)}')
```

predict function uses k nearest neighbor to classify which subject the training and testing image belongs to. The result is listed in the following table:

	Training	Testing
Accuracy	0.8962962962962963	0.9666666666666667

Implement Kernel LDA

```
def make_kernel(images, kernel):
    if kernel == 'rbf':
        similarity = squareform(pdist(images), 'sqeuclidean')
        gram = np.exp(-gamma * similarity)
    elif kernel == 'linear':
        gram = np.dot(images, images.T)
    return gram

def kernel_LDA(train, test, no_dims=25):
    kernels = ['linear', 'rbf']
    for kernel in kernels:
        print('Computing Meam...')
        all_mean = np.mean(train, axis=0).reshape(-1, 1).T
        class_mean = []
        for idx in range(15):
            rows = train[9 * idx:9 * (idx + 1), :]
            class_mean.append(np.mean(rows, axis=0))
        class_mean = np.array(class_mean).astype(np.float32)

        print('Computing SW...')
        (n, d) = train.shape
        temp = train.copy().astype(np.float32)
        for data in range(n):
            temp[data, :] -= class_mean[data // 9, :]
        within_class = make_kernel(temp.T, kernel)

        print('Computing SB...')
        (n, d) = class_mean.shape
        temp = class_mean.copy()
        for data in range(n):
            temp[data, :] -= all_mean[0, :]
        between_class = make_kernel(temp.T, kernel)
        between_class *= 9

        print('Computing Eigenvector...')
        eigen_values, eigen_vectors =
np.linalg.eigh(np.dot(np.linalg.pinv(within_class), between_class))
        print('Computing Principle Component...')
        principle_component = eigen_vectors[:, np.argsort(eigen_values)[::-1]
[0: no_dims]].real.astype(np.float32)

        train_low_images = np.dot(train, principle_component)
```

```
        test_low_images = np.dot(test, principle_component)

        predict_result = predict(train_low_images, train_low_images, 5,
train_label)
        print(
            f'Predict result of Training data set using {kernel} kernel is
{len(predict_result[predict_result == train_label]) / len(train_label)}')
        predict_result = predict(train_low_images, test_low_images, 5,
train_label)
        print(
            f'Predict result of Testing data set using {kernel} kernel is
{len(predict_result[predict_result == test_label]) / len(test_label)}')

kernel_LDA(train_images, test_images)
```

make_kernel function implements two types of kernel, which are RBF kernel and linear kernel. **kernel_LDA** function implements the kernel LDA algorithm and extracts the principle components from the images, then using this principle components to do the dimensionality resuction and compute the performance. The accuracy is listed in the following table:

	Training	**Testing**
linear	0.8962962962962963	0.9666666666666667
rbf	0.9703703703703703	0.9333333333333333

SNE

Modification of Code

The different codes between t-SNE and symmetric SNE are as follow:

- Compute pairwise affinities

```
# Compute pairwise affinities
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)

# for t-SNE
num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
# for symmetric SNE
num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))

num[range(n), range(n)] = 0.
Q = num / np.sum(num)
Q = np.maximum(Q, 1e-12)
```

- Compute gradient

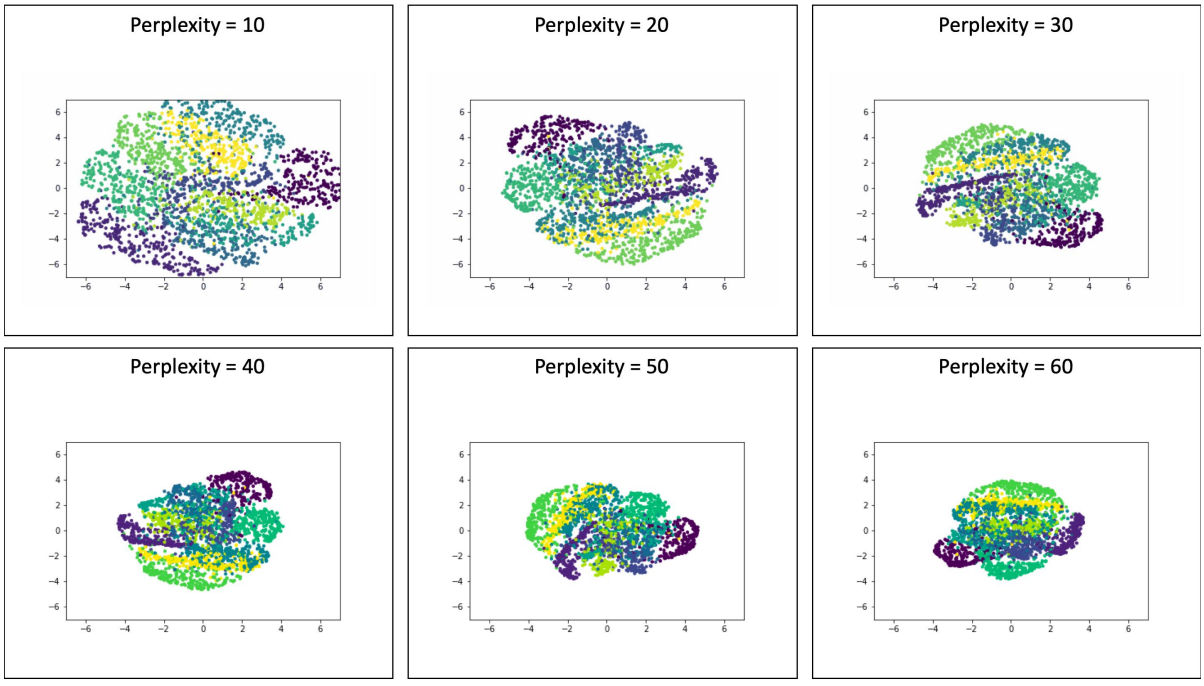
```
# Compute gradient
PQ = P - Q
for i in range(n):
    # for t-SNE
    dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :]
- Y), 0)
    # for symmetric SNE
    dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Visualization of Result

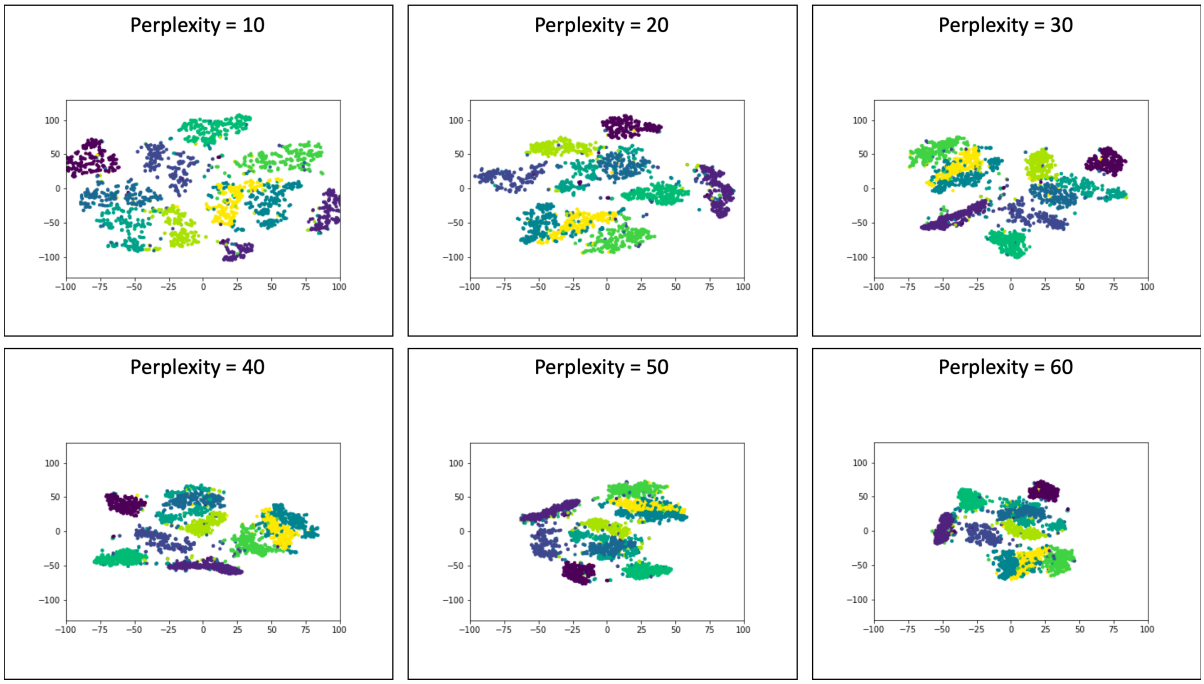
```
def plot_result(result, perplexity, iteration):
    pylab.clf()
    pylab.xlim((-7, 7))
    pylab.ylim((-7, 7))
    pylab.scatter(result[:, 0], result[:, 1], 10, labels)
    pylab.savefig(f'./Result/{method}/{perplexity}_{iteration}.png')
```

plot_result function projects all your data onto 2D space and mark the data points into different colors according to their labels. The following figures show the result:

- Symmetric SNE



- t-SNE

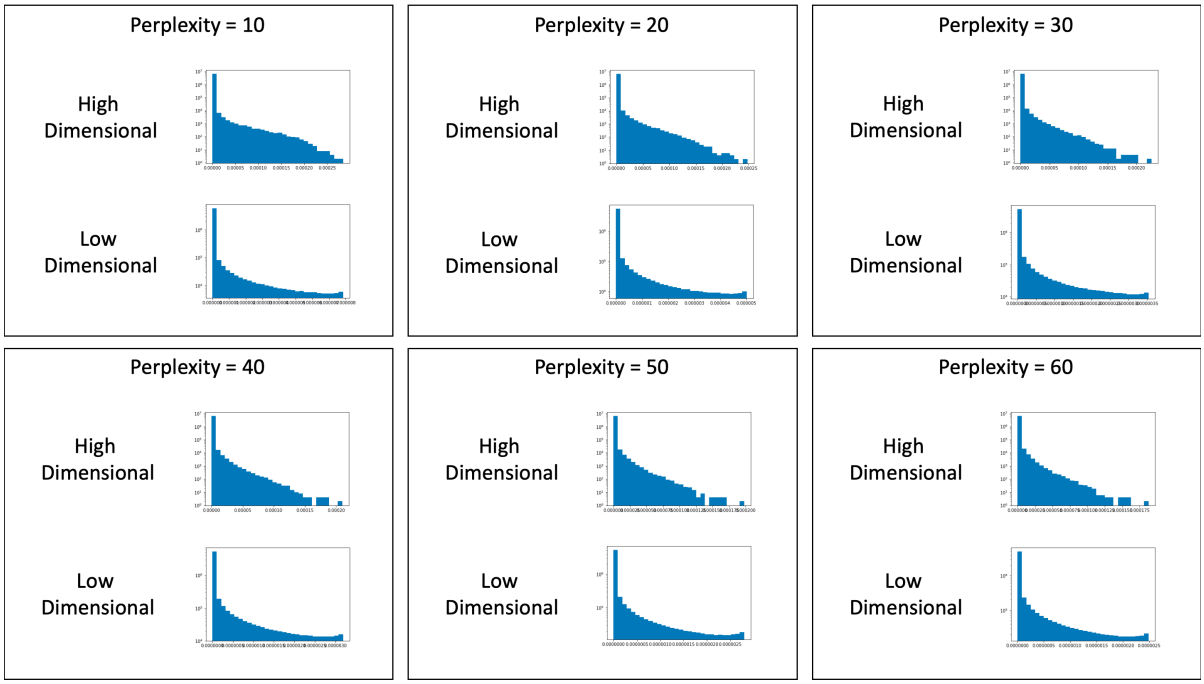


Show Distribution of Pairwise Similarity

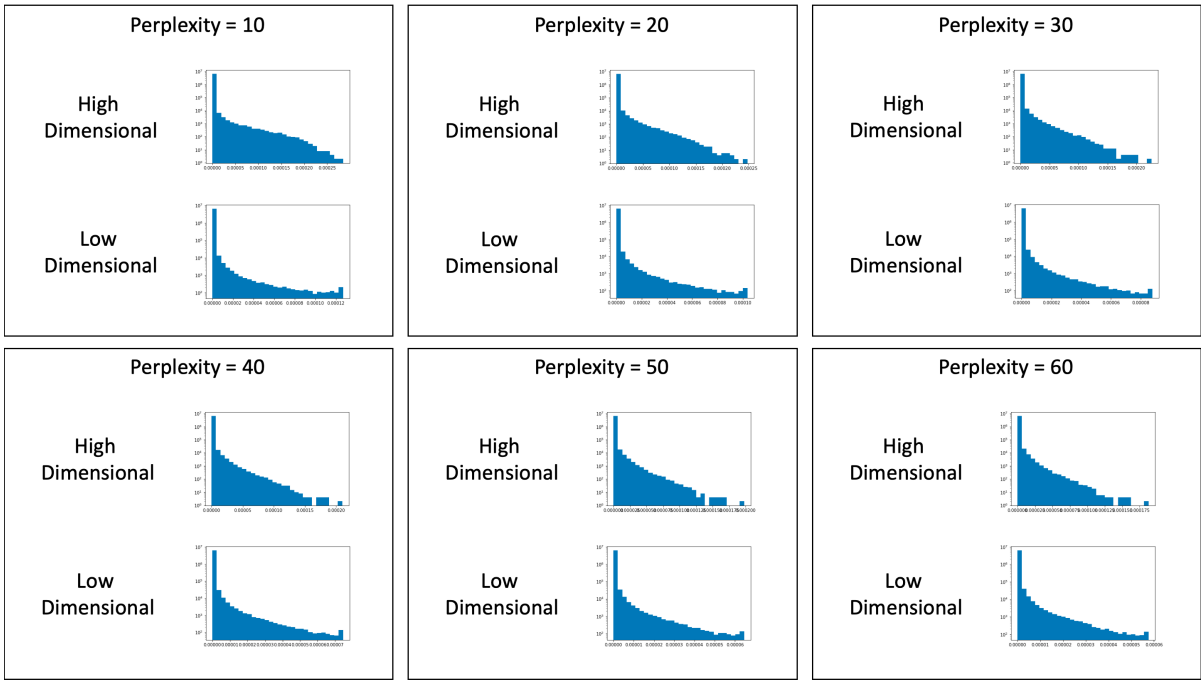
```
def plot_high(Probability, perplexity):
    pylab.clf()
    pylab.hist(Probability.flatten(), bins=30, log=True)
    pylab.savefig(f'./Result/s-SNE/high_{perplexity}.png')

def plot_low(Probability, perplexity):
    pylab.clf()
    pylab.hist(Probability.flatten(), bins=30, log=True)
    pylab.savefig(f'./Result/s-SNE/low_{perplexity}.png')
```

plot_high function visualizes the distribution of pairwise similarities in high-dimensional space.
plot_low function visualizes the distribution of pairwise similarities in low-dimensional space.
The following figures show the result:

- Symmetric SNE



- t-SNE



Discussion

- The different perplexity values I have tried are 10, 20, 30, 40, 50, and 60. As the perplexity value increase, the effect of seperating data points is becoming worse and worse.
- As the figures of result showed, t-SNE can solve the crowding problem which symmetric SNE encountered.
- Because the distribution of pairwise similarities in low-dimensional space is student-t distribution, its bandwidth is much wider than symmetric SNE. The distribution of pairwise similarities in high-dimensional space is the same in t-SNE and symmetric SNE.

Appendix

In this part I will explain my naming principle of result and directory structure.
The directory structure should look like as following:

- Result
  - PCA
    - The figure shows eigenfaces is stored in format of eigenfaces_{number}.png
    - The figure shows randomly chosen images before reconstruction is stored in format of origin_{index_of_image}.png
    - The figure shows randomly chosen images after reconstruction is stored in format of reconstruct_{index_of_image}.png
  - LDA
    - The figure shows fisherfaces is stored in format of fisherfaces_{number}.png
    - The figure shows randomly chosen images before reconstruction is stored in format of origin_{index_of_image}.png
    - The figure shows randomly chosen images after reconstruction is stored in format of reconstruct_{index_of_image}.png
  - s-SNE
    - The GIF shows the optimize procedure is stored in format of s_SNE_{perplexity_value}.gif
    - The figure shows distribution of pairwise similarities in high-dimensional space is stored in format of high_{perplexity_value}.png
    - The figure shows distribution of pairwise similarities in low-dimensional space is stored in format of low_{perplexity_value}.png
  - t-SNE
    - The GIF shows the optimize procedure is stored in format of t_SNE_{perplexity_value}.gif
    - The figure shows distribution of pairwise similarities in high-dimensional space is stored in format of high_{perplexity_value}.png
    - The figure shows distribution of pairwise similarities in low-dimensional space is stored in format of low_{perplexity_value}.png