Machine Learning HW7

1. Code with detailed explanations

This part we are going to explain our code with detailed explanations.

- **Kernel EigenFaces:**
  Part 1:
  In part 1, we need to use PCA and LDA to show the first 25 eigenfaces and fisherfaces, and randomly pick 10 images to show their reconstruction.
  - PCA:
    According to the lecture, we know that the optimal solution is the 25 largest eigenvectors of matrix S.

$$z_1 = w^1 \cdot x$$
$$\left\|w^1\right\| = 1$$
$$\underset{w^1}{\operatorname{argmax}} (w^1)^\top S(w^1) \text{ , subject to } \left\|w^1\right\| = 1$$
$$\text{where } S = \left[\frac{1}{N}\sum_x (x-\bar{x})(x-\bar{x})^\top\right]$$

    Thus, we first construct matrix S in our implementation.

```
train_x = train_dataset.reshape(135,-1).copy()
mean_train_x = np.mean(train_x, axis=0)[:,None] # (c,1)
train_x_transpose = train_x.T
if(kernel == "no"):
    S = (train_x_transpose - mean_train_x) @ (train_x_transpose - mean_train_x).T * (1/train_x.shape[0]) # S (number of data points, channel)
```

    We use the 25 largest eigenvectors to construct our orthogonal matrix.

```
eigenvalues, eigenvectors = np.linalg.eig(S)
sorted_indices = np.argsort(eigenvalues)[::-1]  # Indices of sorted eigenvalues in descending order
top_n_indices = sorted_indices[:n]  # Indices of the top n eigenvalues
top_n_eigenvectors = eigenvectors[:, top_n_indices].real# Get the top n eigenvectors

# We need to store the result of top_n_eigenvector

target_eigenvectors = find_target_eigenvectors(S)

return top_n_eigenvectors
```

    We randomly choose ten images from the training dataset. We use the following equation to reconstruct the result.

```
reconstruct_data = data.reshape(-1) @ eigen_matrix @ eigen_matrix.T
reconstruct_data = reconstruct_data.reshape(29,24) # I misunderstand the width x height
store_data(f"part1/randomly_select_{idx}_{args.analysis}.png",reconstruct_data)
store_data(f"part1/randomly_select_{idx}_{args.analysis}_gt.png",gt_data)
```

  - LDA:
    Actually, LDA and PCA behave similarly in this part, except for the difference in the matrix S. Thus, we would focus on constructing matrix S in this part.
    Based on the lecture, we know that the matrix S is composed of:

$$S_W^{-1} S_B$$

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

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

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

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

We follow this instruction to construct our S.

```
all_mean = np.mean(train_x,axis=0)
class_set = set([x for x in train_label_dataset])
class_mean = []
for c in class_set:
    class_mean.append(np.mean(train_x[train_label_dataset == c],axis=0))
class_mean = np.array(class_mean)
```

All_mean represents the mean of the dataset. class_mean represent the mean of the class.

```
# construct S_b
S_b = np.zeros((train_x.shape[1],train_x.shape[1]))

for idx,c in enumerate(class_set):
    n = np.sum(train_label_dataset == c)
    diff = (class_mean[idx]-all_mean).reshape(train_x.shape[1],1)
    S_b += n * (diff) @ (diff.T)
# construct S_w
S_w = np.zeros((train_x.shape[1],train_x.shape[1]))
for idx,c in enumerate(class_set):
    diff = train_x[train_label_dataset == c] - class_mean[idx]
    S_w += ( diff.T @  diff )

return np.linalg.pinv(S_w) @ S_b
```

Since S_w could be not invertible, we use pseudo inverse to inverse it.

Part 2:

In this part, we need to use PCA and LDA to do face recognition, and compute the performance. We follow the instruction to use knn to determine which subject the testing image belongs to.

First, we use the code of part1 to get the orthogonal matrix. Then, we use this matrix to project our training data and testing data to low dimensions.

```
train_distribution = compute_distribution(eigen_matrix,train_dataset)
test_distribution = compute_distribution(eigen_matrix,test_dataset)
```

```
def compute_distribution(eigen_matrix,train_dataset):
    train_x = train_dataset.reshape(train_dataset.shape[0],-1).copy()
    return train_x @ eigen_matrix
```

After getting distribution, we use the nearest k training data to determine the class of testing data.

```
err = 0
for idx,testcase in enumerate(tqdm(test_distribution)):
    testcase = testcase[None]
    dis = (train_distribution - testcase) * (train_distribution - testcase) # (n,c)
    dis = np.sum(dis,axis=1) #(n)
    smallest_k_idx = np.argsort(dis)[:k]
    bin_class = {}
    for neighbor in train_label_dataset[smallest_k_idx]:
        if(neighbor not in bin_class.keys()):
            bin_class[neighbor] = 0
        else:
            bin_class[neighbor] += 1
    max_key = max(bin_class, key=bin_class.get)
    if(test_label_dataset[idx] != max_key):
        err += 1
print(f"The number of error is: {err}")
print(f"The number of error rate is {err/test_distribution.shape[0]}")
```

Part3:

In this part, we need to use kernel PCA and kernel LDA to do face recognition, and compute the performance. This part is exactly the same as part2, except for the construction of matrix S. Thus, we would focus on the construction of matrix S.

- PCA:
  We follow the instructions of the lecture.

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

```
# Compute kernel
if not kernel_type:
    # Linear
    kernel = training_images.T.dot(training_images)
else:
    # RBF
    kernel = np.exp(-gamma * cdist(training_images.T, training_images.T, 'sqeuclidean'))

# Get centered kernel
matrix_n = np.ones((29 * 24, 29 * 24), dtype=float) / (29 * 24)
matrix = kernel - matrix_n.dot(kernel) - kernel.dot(matrix_n) + matrix_n.dot(kernel).dot(matrix_n)
```

- LDA:
  Based on the Wikipedia, we know that the construction of matrix S would be the following:

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

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

```
kernel_of_each_class = np.zeros((num_of_classes, 29 * 24, 29 * 24))
for idx in range(num_of_classes):
    images = training_images[training_labels == idx + 1]
    kernel_of_each_class[idx] = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
kernel_of_all = np.exp(-gamma * cdist(training_images.T, training_images.T, 'sqeuclidean'))
```

```
matrix_m_i = np.zeros((num_of_classes, 29 * 24))
for idx, kernel in enumerate(kernel_of_each_class):
    for row_idx, row in enumerate(kernel):
        matrix_m_i[idx, row_idx] = np.sum(row) / num_of_each_class[idx]
matrix_m_star = np.zeros(29 * 24)
for idx, row in enumerate(kernel_of_all):
    matrix_m_star[idx] = np.sum(row) / num_of_images
matrix_m = np.zeros((29 * 24, 29 * 24))
for idx, num in enumerate(num_of_each_class):
    difference = (matrix_m_i[idx] - matrix_m_star).reshape((29 * 24, 1))
    matrix_m += num * difference.dot(difference.T)

# Get N^(-1) * M
info_log('=== Calculate inv(N) * M ===')
matrix = np.linalg.pinv(matrix_n).dot(matrix_m)

return matrix
```

- **t-SNE:**

  Part1:

  The main difference between t-SNE and symmetric SNE is the method of representing in low dimensional space.

  - t-SNE:

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

  - SNE:

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

  Thus, we modify the part of q.

```
if not mode:
    # t-SNE
    num = 1. / (1. + np.add(np.add(num, sum_y).T, sum_y))
else:
    # symmetric SNE
    num = np.exp(-1. * np.add(np.add(num, sum_y).T, sum_y))
```

  Also, we need to modify the part calculating gradient.

```
p_q = p - q
if not mode:
    # t-SNE
    for i in range(n):
        d_y[i, :] = np.sum(np.tile(p_q[:, i] * num[:, i], (no_dims, 1)).T * (solution_y[i, :] - solution_y), 0)
else:
    # symmetric SNE
    for i in range(n):
        d_y[i, :] = np.sum(np.tile(p_q[:, i], (no_dims, 1)).T * (solution_y[i, :] - solution_y), 0)
```

  Part2:

  We visualize all the iterations of calculating the final result and represent it with .gif files.

```
# Compute current value of cost function
if (iteration + 1) % 10 == 0:
    c = np.sum(p * np.log(p / q))
    info_log(f'Iteration {iteration + 1}: error is {c}...')
    img.append(capture_current_state(solution_y, labels, mode, perplexity))
```

```
def capture_current_state(solution_y: np.ndarray, labels: np.ndarray, mode: int, perplexity:
    """
    Capture current state
    :param solution_y: current solution
    :param labels: labels of images
    :param mode: 0 for t-SNE, 1 for symmetric SNE
    :param perplexity: perplexity
    :return: an image of current state
    """
    plt.clf()
    plt.scatter(solution_y[:, 0], solution_y[:, 1], 20, labels)
    plt.title(f'{"t-SNE" if not mode else "symmetric SNE"}, perplexity = {perplexity}')
    plt.tight_layout()
    canvas = plt.get_current_fig_manager().canvas
    canvas.draw()

    return Image.frombytes('RGB', canvas.get_width_height(), canvas.tostring_rgb())
```

Part3:

We visualize the pairwise similarity in high dimension and low dimension. The p variable in original source code represent the pairwise similarity of data points in high dimension. The q variable in the original source code represent the pairwise similarity of data points in low dimension.Thus, All we need to do is to visualize variables q and p.

```
def draw_similarities(p: np.ndarray, q: np.ndarray, labels: np.ndarray) -> None:
    """
    Draw similarities
    :param p: p
    :param q: q
    :param labels: labels
    :return: None
    """
    # Get sorted index
    index = np.argsort(labels)
    plt.clf()
    plt.figure(1)

    # Plot p
    log_p = np.log(p)
    sorted_p = log_p[index][:, index]
    plt.subplot(121)
    img = plt.imshow(sorted_p, cmap='gray', vmin=np.min(log_p), vmax=np.max(log_p))
    plt.colorbar(img)
    plt.title('High-dimensional space')

    # Plot q
    log_q = np.log(q)
    sorted_q = log_q[index][:, index]
    plt.subplot(122)
    img = plt.imshow(sorted_q, cmap='gray', vmin=np.min(log_q), vmax=np.max(log_q))
    plt.colorbar(img)
    plt.title('Low-dimensional space')

    plt.tight_layout()
```

Part4:

In this part, what we need to do is to modify the coefficient of function and we can get the result.

```
def sne(images: np.ndarray, labels: np.ndarray, mode: int, no_dims: int = 2, initial_dims: int = 50,
        perplexity: float = 20.0) -> np.ndarray:
```

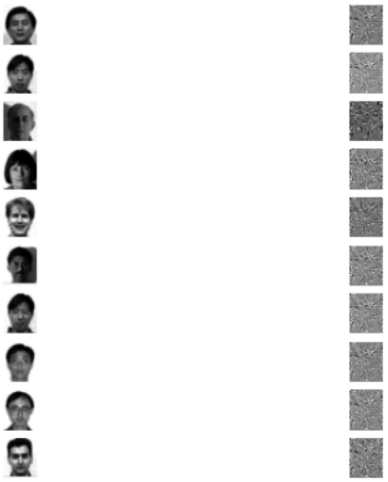
2. Experiments and Discussion

- **Kernel EigenFaces:**

Part1:

PCA:

LDA:



Part2:

PCA:

```
Error count: 4
Error rate: 0.13333333333333333
```

LDA:

```
Error count: 1
Error rate: 0.03333333333333333
```

Part3:

PCA:

We use error rate as our metric to compare performance of different

settings.

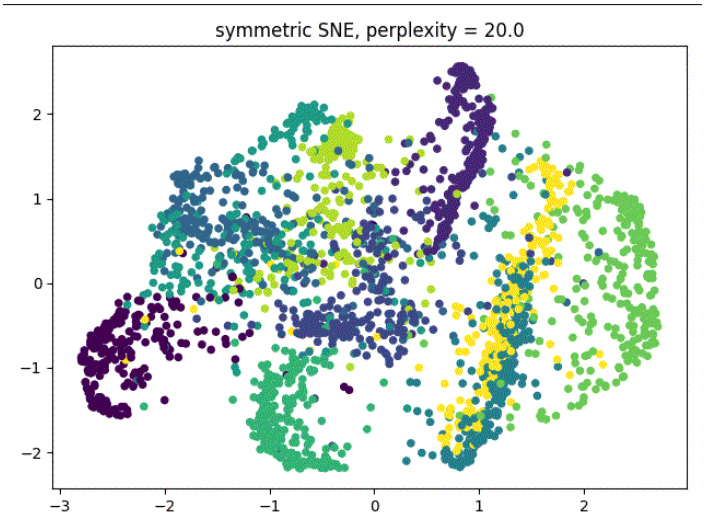
	Error rate
PCA - without kernel	13%
PCA - Linear kernel	13%
PCA - RBF	13%

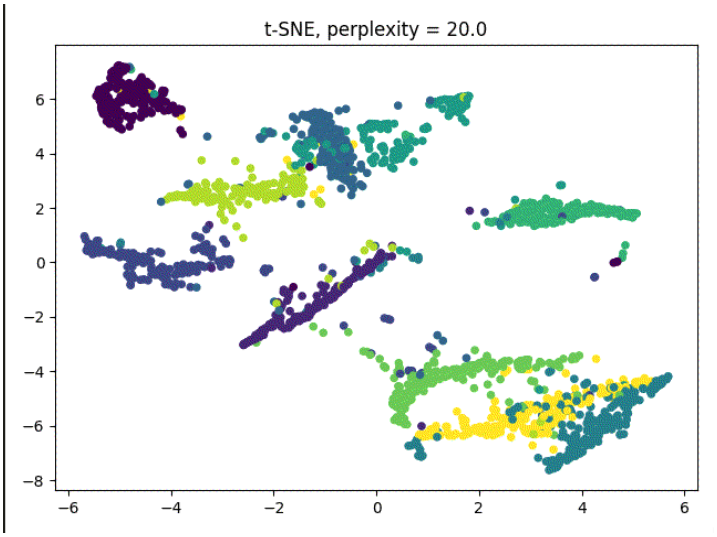
LDA:

	Error rate
PCA - without kernel	3%
PCA - Linear kernel	80%
PCA - RBF	27%

We can find that PCA without a kernel gets the best performance. I think the reason is that the original data points are already well separated in the high-dimensional space, so projecting them into the feature space might result in a less optimal distribution.

- **t-SNE:**
  Part1:





As we can see, t-sne uses long tail distribution to represent data points in low dimension. Thus, it does not have crowd problems compared with sne.

Part2:

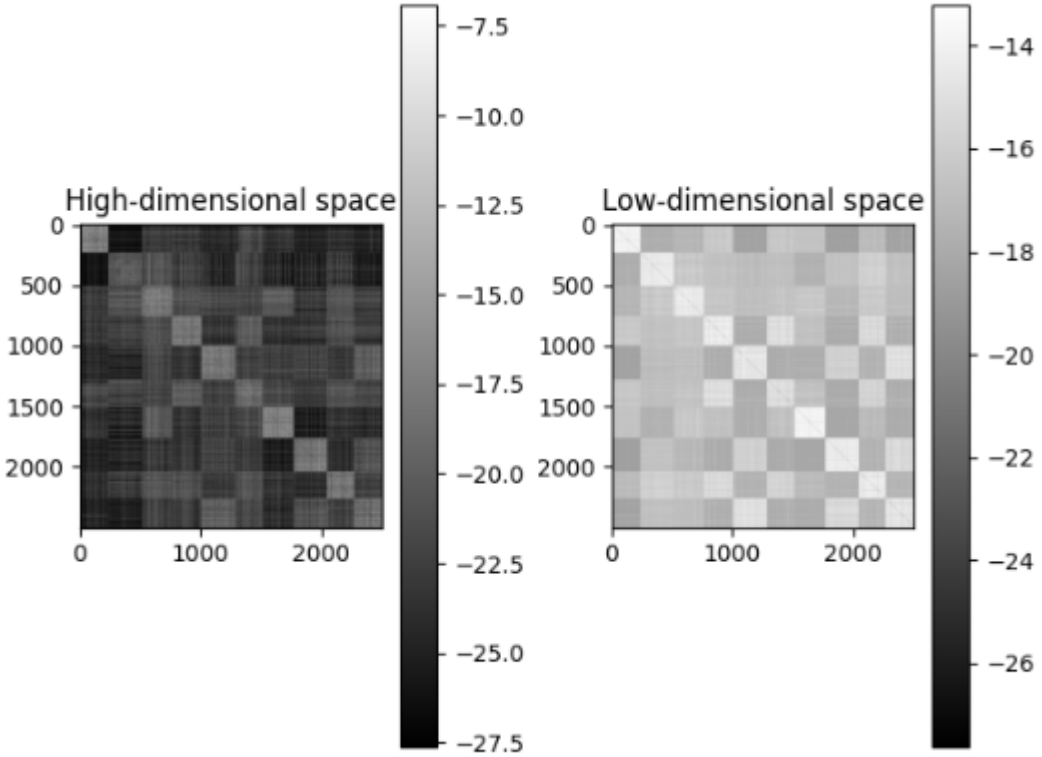
The visualization result is in here:

(t-sne): Link

(sne): Link

Part3:

(t-sne):



(sne):

Part4:

(t-sne):

Perplexity=20 -> Link

Perplexity=30 -> Link
Perplexity=40 -> Link

(sne):

Perplexity=20 -> Link
Perplexity=30 -> Link
Perplexity=40 -> Link

Perplexity will affect the number of neighbors to be considered. The larger The perplexity is, the less sensitive the points are to smaller groups. Because symmetric SNE suffers from crowded problem, the effect of perplexity is not clear. However, the effect of perplexity is clear in t-SNE as it gets larger.

3. Observations and Discussion

Here are some observations I made while completing this homework.

- Reconstructed faces from LDA are not as clear as those from PCA. However, the performance of LDA is far better than PCA. I think computer facial recognition is quite different from that of humans.
- Eigenfaces depict the contour of the faces so as to extract the features of the images. Based on these features, we can classify the testing images.
- From the scatter graphs, symmetric SNE suffers from the crowded problem. It is hard to distinguish different classes without color. Nevertheless, t-SNE uses t-distribution to alleviate crowded problems. Points far away from each other in high-dimensional space still have far distance from each other in low-dimensional space.
- The speed of convergence of symmetric SNE is faster than that of t-SNE via the results of gif files.