

ML-HW7

312657008_江祐姍

1. Code with detailed explanations

o Kernel Eigenfaces

[load data]

```
# train_load
train_path = "C:\\Users\\yoush\\Desktop\\機器學習\\HW7\\Yale_Face_Database\\Yale_Face_Database\\Training\\"
files = os.listdir(train_path)

train_image = []
train_subclass = 9
train_label = [[i] * train_subclass for i in range(15)]

for file in files:

    file_path = os.path.join(train_path, file)
    image = Image.open(file_path)
    train_image.append(np.array(image).reshape(-1))

train_image = np.array(train_image)
train_label = np.array(train_label).reshape(-1)

# test_load
test_path = "C:\\Users\\yoush\\Desktop\\機器學習\\HW7\\Yale_Face_Database\\Yale_Face_Database\\Testing\\"
files = os.listdir(test_path)

test_image = []
test_subclass = 2
test_label = [[i] * test_subclass for i in range(15)]

for file in files:
```

```

file_path = os.path.join(test_path, file)
image = Image.open(file_path)
test_image.append(np.array(image).reshape(-1))

test_image = np.array(test_image)
test_label = np.array(test_label).reshape(-1)

```

Part1

```

def PCA(train_image, train_label):
    mean = np.mean(train_image, axis=0)
    center = train_image - mean

    S = np.cov(train_image, bias=True)
    eigval, eigvec = np.linalg.eig(S)
    index = np.argsort(eigval)[::-1]
    eigval = eigval[index].real
    eigvec = eigvec[:, index].real

    # first 25 eigenfaces
    transform = center.T @ eigvec
    fig, axes = pyplot.subplots(5, 5, figsize=(10, 10))

    for i, ax in enumerate(axes.flat):
        ax.axis("off")
        ax.imshow(transform[:, i].reshape(231, 195), cmap="gray")
    pyplot.show()

    # reconstruct
    z = transform.T @ center.T
    reconstruct = transform @ z + mean.reshape(-1, 1)
    index = np.random.choice(135, 10, replace=False) # 9*15=135

    fig, axes = pyplot.subplots(2, 10, figsize=(15, 6))
    for i, ax in enumerate(axes.flat):
        ax.axis("off")
        if i < 10:

```

```

        ax.imshow(train_image[index][i].reshape(231, 195), cmap="gray")
    else:
        ax.imshow(reconstruct[:, index][:, i - 10].reshape(231, 195),
cmap="gray")
    pyplot.show()

    return transform, z

def LDA(pca_transform, pca_z, train_image, train_label):

    mean = np.mean(pca_z, axis=1)
    N = pca_z.shape[0]

    S_within = np.zeros((N, N))
    for i in range(15):
        S_within += np.cov(pca_z[:, i*9:i*9+9], bias=True)

    S_between = np.zeros((N, N))
    for i in range(15):
        class_mean = np.mean(pca_z[:, i*9:i*9+9], axis=1).T
        S_between += 9 * (class_mean - mean) @ (class_mean - mean).T

    S = np.linalg.inv(S_within) @ S_between
    eigval, eigvec = np.linalg.eig(S)
    index = np.argsort(eigval)[::-1]
    eigval= eigval[index].real
    eigvec = eigvec[:,index].real

    # first 25 fisherfaces
    transform = pca_transform @ eigvec
    fig, axes = pyplot.subplots(5, 5, figsize=(10, 10))

    for i, ax in enumerate(axes.flat):
        ax.axis("off")
        ax.imshow(transform[:, i].reshape(231, 195), cmap="gray")
    pyplot.show()
    mean = np.mean(train_image, axis=0)

```

```

center = train_image - mean
z = transform.T @ center.T

# reconstruct
reconstruct = transform @ z + mean.reshape(-1, 1)
fig, axes = pyplot.subplots(2, 10, figsize=(15, 6))
for i, ax in enumerate(axes.flat):
    ax.axis("off")
    if i < 10:
        ax.imshow(train_image[index][i].reshape(231, 195), cmap="gray")
    else:
        ax.imshow(reconstruct[:, index][:, i - 10].reshape(231, 195),
cmap="gray")
pyplot.show()

fisher_z = eigvec.T @ pca_z
return fisher_z, transform

pca_transform, pca_z = PCA(train_image, train_label)
fisher_z, fisher_transform = LDA(pca_transform, pca_z, train_image,
train_label)

```

● Principal Component Analysis (PCA)

Steps:

1. **Input:** The function PCA takes train_image (image dataset) and train_label (class labels) as inputs.
2. **Step 1: Compute the Mean**
 - Compute the mean image vector across all samples (mean).
 - mean has the same dimensions as an image.
3. **Step 2: Center the Data**
 - Subtract the mean image vector from each image.
 - center represents the mean-centered dataset.

4. **Step 3: Compute Covariance Matrix**

- Compute the covariance matrix (S) of the dataset.

5. **Step 4: Eigen Decomposition**

- Compute eigenvalues (eigval) and eigenvectors (eigvec) of the covariance matrix.

6. **Step 5: Sort Eigenvalues and Eigenvectors**

- Sort eigenvalues in descending order and reorder eigenvectors accordingly.

7. **Step 6: Visualize the First 25 Eigenfaces**

- Project the centered images onto the eigenvector space to obtain eigenfaces.
- Visualize the first 25 eigenfaces in a 5x5 grid.

8. **Step 7: Reconstruct Images**

- Compute the projections (z) of the centered images in the PCA space.
- Reconstruct the original images from the PCA projections.
- Visualize original images alongside their reconstructions.

9. **Output:**

- Returns the transformation matrix (transform) and projections (z) in the PCA space.

● **Linear Discriminant Analysis (LDA)**

Steps:

1. **Input:** The function LDA takes the PCA results (pca_transform and pca_z) and the original data (train_image and train_label) as inputs.

2. **Step 1: Compute Mean in PCA Space**

- Compute the mean vector of all PCA projections.mean has the same

dimensions as an image.

3. **Step 2: Compute Within-Class Scatter Matrix**

- Compute the scatter within each class and sum them to form S_{within} .
- Each class has 9 samples, iterated across 15 classes.

4. **Step 3: Compute Between-Class Scatter Matrix**

- Compute the scatter between classes using their means.
- Multiply by the number of samples per class (9).

5. **Step 4: Solve Generalized Eigenvalue Problem**

- Solve for the eigenvalues and eigenvectors of $S = \text{inv}(S_{\text{within}}) @ S_{\text{between}}$.

6. **Step 5: Sort Eigenvalues and Eigenvectors**

- Sort eigenvalues in descending order and reorder eigenvectors accordingly.

7. **Step 6: Visualize the First 25 Fisherfaces**

- Transform the PCA results into the LDA space to obtain Fisherfaces.
- Visualize the first 25 Fisherfaces.

8. **Step 7: Reconstruct Images**

- Compute projections in the LDA space.
- Reconstruct original images using LDA projections.
- Visualize original images alongside their reconstructions.

9. **Output:**

- Returns the LDA-transformed projections (fisher_z) and the Fisherface transformation matrix (transform).

Part2

```
def face_recognition(fisher_z, fisher_transform, train_image,
train_label, test_image, test_label, k=3):

    train_proj = fisher_z.T

    mean = np.mean(train_image, axis=0)
    test_centered = test_image - mean
    test_proj = fisher_transform.T @ test_centered.T

    # k-NN
    distances = cdist(test_proj.T, train_proj, metric="euclidean")
    neighbors = np.argsort(distances, axis=1)[: , :k]
    pred_labels = []

    for i in range(test_proj.shape[1]):
        neighbor_labels = train_label[neighbors[i]]
        pred_labels.append(np.argmax(np.bincount(neighbor_labels)))

    correct_count = sum([1 for i in range(len(test_label)) if
test_label[i] == pred_labels[i]])
    accuracy = correct_count / len(test_label)
    print(f"Accuracy: {accuracy * 100:.2f}%")
    return pred_labels

predicted_labels = face_recognition(fisher_z, fisher_transform,
train_image, train_label, test_image, test_label, k=3)
```

The `face_recognition` function performs face recognition using features extracted through PCA (Principal Component Analysis) and LDA (Linear Discriminant Analysis). It uses the k-Nearest Neighbors (k-NN) algorithm for classification. Here's how the function works:

1. Project Training Data

- The input `fisher_z` represents the LDA-transformed feature vectors for the training data.
- The transpose (`fisher_z.T`) changes the shape so that rows represent training samples and columns represent features. This is necessary for distance-based classification.

2. Center Test Images

- The mean of all training images is calculated pixel-wise (i.e., for each pixel location across all training samples). This is the mean image.
- The `test_image` data (test set) is then centered by subtracting this mean. This ensures the test data is aligned with the training data (mean-centered) in the same space.

3. Project Test Data into Fisher Space

- The LDA transformation (`fisher_transform`) is applied to the centered test data:
 - `fisher_transform.T` (transpose of the Fisher transformation matrix) is used to project the test data.
 - The test data is transposed (`test_centered.T`) so the projection works as a matrix multiplication.
- The resulting `test_proj` contains the LDA feature representation of the test data in the same subspace as the training data.

4. Compute Pairwise Distances

- Euclidean distances are computed between every test sample (rows of `test_proj.T`) and every training sample (rows of `train_proj`):
 - `cdist` (from `scipy.spatial.distance`) calculates the pairwise distances efficiently.
 - `distances` is a 2D matrix where each row corresponds to a test sample and each column corresponds to a training sample.

5. Find k-Nearest Neighbors

- The distances are sorted for each test sample along the second axis (columns) using `np.argsort`.
- The indices of the top k smallest distances (nearest neighbors) are selected for each test sample. This results in the neighbors matrix:
 - Each row contains the indices of the k nearest training samples for a test sample.

6. Assign Labels Using Majority Voting

- For each test sample:
 - The labels of its k nearest neighbors are retrieved from `train_label` using the indices in `neighbors[i]`.
 - Majority Voting is applied:
 - `np.bincount` counts the occurrences of each label among the k neighbors.
 - `np.argmax` selects the label with the highest count (most frequent).
- The predicted label for each test sample is stored in the `pred_labels` list.

7. Compute Accuracy

- The number of correctly classified test samples is counted:
 - A comparison is made between the true labels (`test_label`) and the predicted labels (`pred_labels`).
 - If a predicted label matches the true label, it's considered correct.
- The accuracy is calculated as the ratio of correctly classified samples to the total number of test samples.

8. Print Accuracy

- The accuracy is displayed as a percentage, rounded to two decimal

places.

9. Return Predicted Labels

- The function returns the predicted labels for all test samples.

Part3

```
def kernel_PCA(train_image, train_label, kernel, k=3):

    mean = np.mean(train_image, axis=0)
    center = train_image - mean

    N = train_image.shape[0]
    if kernel == "linear":
        K = train_image @ train_image.T
    elif kernel == "poly":
        K = (train_image @ train_image.T) ** 3
    elif kernel == "RBF":
        K = np.exp(-0.01 * cdist(train_image, train_image, 'sqeuclidean'))

    one_N = np.ones((N, N)) / N
    K_centered = K - one_N @ K - K @ one_N + one_N @ K @ one_N

    eigval, eigvec = np.linalg.eig(K_centered)
    index = np.argsort(eigval)[::-1]
    eigval = eigval[index].real
    eigvec = eigvec[:, index].real

    transform = center.T @ eigvec

    z = transform.T @ center.T

    return transform, z

def kernel_LDA(pca_transform, pca_z, train_image, train_label):
```

```

mean = np.mean(pca_z, axis=1)
N = pca_z.shape[0]

S_within = np.zeros((N, N))
for i in range(15):
    S_within += np.cov(pca_z[:, i*9:i*9+9], bias=True)

S_between = np.zeros((N, N))
for i in range(15):
    class_mean = np.mean(pca_z[:, i*9:i*9+9], axis=1).T
    S_between += 9 * (class_mean - mean) @ (class_mean - mean).T

S = np.linalg.inv(S_within) @ S_between
eigval, eigvec = np.linalg.eig(S)
index = np.argsort(eigval)[::-1]
eigval = eigval[index].real
eigvec = eigvec[:, index].real

fisher_transform = pca_transform @ eigvec

fisher_z = eigvec.T @ pca_z

return fisher_z, fisher_transform

def face_recognition(fisher_z, fisher_transform, train_image, train_label,
test_image, test_label, kernel, k=3):

    train_proj = fisher_z.T
    mean = np.mean(train_image, axis=0)
    test_centered = test_image - mean
    test_proj = fisher_transform.T @ test_centered.T

    distances = cdist(test_proj.T, train_proj, metric="euclidean")

```

```

neighbors = np.argsort(distances, axis=1)[: , :k]
pred_labels = []

for i in range(test_proj.shape[1]):
    neighbor_labels = train_label[neighbors[i]]
    pred_labels.append(np.argmax(np.bincount(neighbor_labels)))

correct_count = sum([1 for i in range(len(test_label)) if test_label[i] ==
pred_labels[i]])
accuracy = correct_count / len(test_label)
print(f"Kernel: {kernel} | Accuracy: {accuracy * 100:.2f}%")

return pred_labels

```

The function performs dimensionality reduction using kernel PCA. It accepts training images, their labels, a kernel type, and optionally the number of components (k).

Kernel PCA Function: kernel_PCA

Steps:

1. Center the Data

- Subtract the mean of the training images to center the data.

2. Construct the Kernel Matrix K

- Depending on the kernel type
 - Linear Kernel

$$K = XX^T \text{ where } X \text{ is the training data.}$$

- Polynomial Kernel

$$K = (XX^T)^3 \text{ where } X \text{ is the training data.}$$

- RBF Kernel

$$K = e^{-\gamma \cdot \text{squclidean}(X,X)}$$

Use cdist for pairwise squared Euclidean distance.

3. Center the Kernel Matrix

Kernel PCA requires centering the kernel matrix

$$K_{centered} = K - \frac{1}{N} \mathbf{1}K - K \frac{1}{N} \mathbf{1} + \frac{1}{N} \mathbf{1}K \frac{1}{N} \mathbf{1}, \mathbf{1} \text{ is an } N * N \text{ matrix of ones.}$$

4. Eigen Decomposition

- Compute eigenvalues and eigenvectors of the centered kernel matrix

$$K_{centered}$$

- Sort them in descending order of eigenvalues.

5. Transform the Data

- Compute the transformed features using eigenvectors.

6. Output

- transform: Principal components.
- z: Projected data in the PCA space.

Kernel LDA Function: kernel_LDA

The function performs Linear Discriminant Analysis in the PCA-transformed space to maximize class separability.

Steps

1. Compute Class Means

- Calculate the mean vector for each class and the global mean.

2. Within-Class Scatter Matrix (S_within)

- Sum the covariance of each class, weighted by the number of samples per class.

3. Between-Class Scatter Matrix ($S_{between}$)

- Measure the scatter of the class means relative to the global mean.

$$S_{between} = \sum_i N_i (\mu_i - \mu) (\mu_i - \mu)^T$$

4. Solve the Generalized Eigenproblem

- Compute eigenvalues and eigenvectors of

$$S = S_{within}^{-1} S_{between}$$

- Sort eigenvalues and eigenvectors in descending order.

5. Transform Data

- Compute Fisher's discriminant transformation

6. Output

- fisher_z: LDA-transformed data.
- fisher_transform: Transformation matrix.

Face Recognition Function: face_recognition

This function performs classification using the k-NN algorithm in the LDA-transformed space.

Steps

1. Train Projection

Project training data to the Fisher-LDA space.

2. Test Projection

Center the test images and project them using the Fisher-LDA transformation matrix

3. k-Nearest Neighbors

- Compute the Euclidean distance between test samples and all training samples.

- Find the indices of the k nearest neighbors.

4. Majority Voting

- For each test sample, predict the label based on the majority vote of its k neighbors.

5. Calculate Accuracy

- Compare predicted labels with ground truth.

6. Output

- Predicted labels for the test images.
- Prints classification accuracy.

Kernel PCA, Kernel LDA, and Face Recognition for Each Kernel

The script loops over three kernels (linear, poly, RBF), and for each

1. Performs kernel PCA.
2. Applies kernel LDA on PCA-transformed data.
3. Classifies test data using k-NN and reports the accuracy.

o t-SNE

```
def symmetric_sne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0):  
  
    # Check inputs  
    if isinstance(no_dims, float):  
        print("Error: array X should have type float.")  
        return -1  
    if round(no_dims) != no_dims:  
        print("Error: number of dimensions should be an integer.")  
        return -1
```

```

# Initialize variables
X = pca(X, initial_dims).real
(n, d) = X.shape
max_iter = 1000
initial_momentum = 0.5
final_momentum = 0.8
eta = 500
min_gain = 0.01
Y = np.random.randn(n, no_dims)
Y_history = [(0, Y.copy())]
dY = np.zeros((n, no_dims))
iY = np.zeros((n, no_dims))
gains = np.ones((n, no_dims))

# Compute P-values
P = x2p(X, 1e-5, perplexity)
P = P + np.transpose(P)
P = P / np.sum(P)
P = P * 4.          # early exaggeration
P = np.maximum(P, 1e-12)

# Run iterations
for iter in range(max_iter):

    # Compute pairwise affinities
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    num = 1. / (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
    PQ = P - Q
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)

```



```

# Perform the update
if iter < 20:
    momentum = initial_momentum
else:
    momentum = final_momentum

gains = (gains + 0.2) * ((dY > 0.) != (iY > 0.)) + \
        (gains * 0.8) * ((dY > 0.) == (iY > 0.))

gains[gains < min_gain] = min_gain
iY = momentum * iY - eta * (gains * dY)
Y = Y + iY
Y = Y - np.tile(np.mean(Y, 0), (n, 1))
Y_history.append((iter + 1, Y.copy()))

# Compute current value of cost function
if (iter + 1) % 10 == 0:

    C = np.sum(P * np.log(P / Q))
    print("Iteration %d: error is %f" % (iter + 1, C))

# Stop lying about P-values
if iter == 100:
    P = P / 4.

pylab.title('Symmetric-SNE high-dim')
pylab.hist(P.flatten(),bins=40,log=True, color='skyblue')
pylab.show()

pylab.title('Symmetric-SNE low-dim')
pylab.hist(Q.flatten(),bins=40,log=True, color='red')
pylab.show()

# Return solution

pylab.title('Symmetric-SNE')
pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
pylab.show()

return Y, Y_history
def tsne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0):
    """
    Runs t-SNE on the dataset in the Nx D array X to reduce its

```

```

        dimensionality to no_dims dimensions. The syntaxis of the function is
        `Y = tsne.tsne(X, no_dims, perplexity)`, where X is an NxD NumPy array.
    """

    # Check inputs
    if isinstance(no_dims, float):
        print("Error: array X should have type float.")
        return -1
    if round(no_dims) != no_dims:
        print("Error: number of dimensions should be an integer.")
        return -1

    # Initialize variables
    X = pca(X, initial_dims).real
    (n, d) = X.shape
    max_iter = 1000
    initial_momentum = 0.5
    final_momentum = 0.8
    eta = 500
    min_gain = 0.01
    Y = np.random.randn(n, no_dims)
    Y_history = [(0, Y.copy())]
    dY = np.zeros((n, no_dims))
    iY = np.zeros((n, no_dims))
    gains = np.ones((n, no_dims))

    # Compute P-values
    P = x2p(X, 1e-5, perplexity)
    P = P + np.transpose(P)
    P = P / np.sum(P)
    P = P * 4.          # early exaggeration
    P = np.maximum(P, 1e-12)

    # Run iterations
    for iter in range(max_iter):

        # Compute pairwise affinities
        sum_Y = np.sum(np.square(Y), 1)

```

```

num = -2. * np.dot(Y, Y.T)
num = 1. / (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
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)

# Perform the update
if iter < 20:
    momentum = initial_momentum
else:
    momentum = final_momentum
gains = (gains + 0.2) * ((dY > 0.) != (iY > 0.)) + \
    (gains * 0.8) * ((dY > 0.) == (iY > 0.))
gains[gains < min_gain] = min_gain
iY = momentum * iY - eta * (gains * dY)
Y = Y + iY
Y = Y - np.tile(np.mean(Y, 0), (n, 1))
Y_history.append((iter + 1, Y.copy()))
# Compute current value of cost function
if (iter + 1) % 10 == 0:
    C = np.sum(P * np.log(P / Q))
    print("Iteration %d: error is %f" % (iter + 1, C))

# Stop lying about P-values
if iter == 100:
    P = P / 4.

pylab.title('t-SNE high-dim')
pylab.hist(P.flatten(),bins=40,log=True, color='skyblue')
pylab.show()

pylab.title('t-SNE low-dim')

```

```

pylab.hist(Q.flatten(),bins=40,log=True, color='red')
pylab.show()

pylab.title('t-SNE')
pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
pylab.show()

return Y, Y_history

def create_animation(Y_history, labels, title, save_dir, filename, axis_range=(-50,
50)):
    # Ensure the directory exists
    os.makedirs(save_dir, exist_ok=True)
    save_path = os.path.join(save_dir, filename)

    # Create the figure and axis
    fig, ax = plt.subplots(figsize=(10, 10))
    scatter = ax.scatter([], [], c=[], cmap='tab10', s=20)

    # Set axis properties
    ax.set_title(title)
    ax.set_xlim(axis_range)
    ax.set_ylim(axis_range)
    ax.grid(True, linestyle='--', alpha=0.7)
    ax.axhline(y=0, color='k', linestyle='-', alpha=0.3)
    ax.axvline(x=0, color='k', linestyle='-', alpha=0.3)

    # Initialization function
    def init():
        scatter.set_offsets(np.c_[[], []]) # Clear points initially
        scatter.set_array([]) # Reset the color array
        return [scatter] # Return as a list to make it iterable

    # Update function for each frame
    def update(frame):
        iteration, Y = Y_history[frame] # Unpack the frame data
        scatter.set_offsets(np.c_[Y[:, 0], Y[:, 1]]) # Set new positions

```

```

scatter.set_array(labels) # Update the color array based on labels
return [scatter] # Return as a list to make it iterable

# Create the animation object, only for the first 100 frames
frames_to_use = min(100, len(Y_history)) # Ensure it doesn't exceed the length
of Y_history

anim = FuncAnimation(fig, update, init_func=init,
                    frames=frames_to_use,
                    interval=200, # Frame interval in ms
                    blit=True)

# Save the animation to the specified file
writer = PillowWriter(fps=30)
anim.save(save_path, writer=writer)

plt.close() # Close the plot window after saving

print(f"Animation saved to: {save_path}")
print(f"Total frames: {frames_to_use}")

```

Part1

● Symmetric SNE

Gradient update equation

$dY[i, :] = \text{np.sum}(\text{np.tile}(PQ[:, i], (\text{no_dims}, 1))).T * (Y[i, :] - Y), 0)$

- Low-Dimensional Similarity Distribution Q_{ij}

In Symmetric SNE, Q_{ij} is computed using a Gaussian kernel

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

This Gaussian kernel treats the similarity between points in the low-dimensional space as the same form as the high-dimensional similarity.

- Gradient Computation

The gradient is computed based on the difference between P_{ij} and Q_{ij}

$$\frac{\partial C}{\partial y_i} = \sum_j (P_{ij} - Q_{ij})(y_i - y_j)$$

The term $PQ[:, i]$ represents $P_{ij} - Q_{ij}$ for all points relative to point i .

● t-SNE

Gradient update equation

$$dY[i, :] = \text{np.sum}(\text{np.tile}(PQ[:, i] * \text{num}[:, i], (\text{no_dims}, 1)).T * (Y[i, :] - Y), 0)$$

- Low-Dimensional Similarity Distribution Q_{ij}

In t-SNE, Q_{ij} is computed using a Student-t distribution with one degree of freedom (a heavy-tailed distribution)

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

This distribution allows points that are far apart in the low-dimensional space to have non-zero similarity, mitigating the crowding problem.

- Gradient Computation

The gradient in t-SNE includes an additional weighting term $\text{num}[:, i]$ derived from the Student-t distribution

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (P_{ij} - Q_{ij}) Q_{ij} (y_i - y_j)$$

The additional term Q_{ij} (stored in $\text{num}[:, i]$) scales the gradient, emphasizing the influence of nearby points in the low-dimensional space.

Part2

The create_animation function to create gif

1. Setup and Initialization:

- A directory is created (if it doesn't already exist) to save the animation file.
- A matplotlib figure and scatter plot are initialized, with axis limits and gridlines set for clarity.

2. Frames in the Animation:

- The Y_history variable, which stores the low-dimensional embeddings (Y) at each iteration, is used to update the scatter plot in each frame.
- Each frame corresponds to an iteration, showing the positions of all data points in 2D space.

3. Color Coding:

- The labels array determines the color of each data point in the scatter plot, ensuring that different categories are visually distinguishable.

4. Animation:

- The FuncAnimation class from matplotlib.animation is used to create the animation. It updates the scatter plot for each frame, corresponding to different iterations of the optimization.
- The animation is saved as a GIF file using the PillowWriter.

5. Customization:

- The number of frames is limited to the first 100 (or fewer, if Y_history contains fewer entries).
- The axis range and title can be adjusted, and the frame interval (200 milliseconds) determines the speed of the animation.

6. Output:

- After generating the animation, the GIF file is saved to the specified location (save_dir and filename) and a confirmation is printed.

And, `pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)` creates the final cluster.

Part3

P (High-Dimensional Similarities)

P_{ij} is derived using a Gaussian kernel centered at each data point

$$P_{ij} = \frac{\exp(-\|x_i - x_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|x_i - x_k\|^2 / 2\sigma_i^2)}$$

where:

- x_i and x_j are high-dimensional data points.
- σ_i is the bandwidth (determined by the perplexity).

Then, using `hist` to display `P.flatten()` distribution.

Q (Low-Dimensional Similarities)

Q_{ij} is derived using a Student-t distribution (with 1 degree of freedom) instead of a Gaussian

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

where:

- y_i and y_j are low-dimensional data points.
- The Student-t distribution ensures that distances are more spread out in the low-dimensional space, reducing the crowding problem.

Then, using `hist` to display `Q.flatten()` distribution.

Part4

Modify the perplexity values in the `symmetric_sne` and `tsne` functions, and then evaluate the results.



2. Experiments and Discussion

o Kernel Eigenfaces

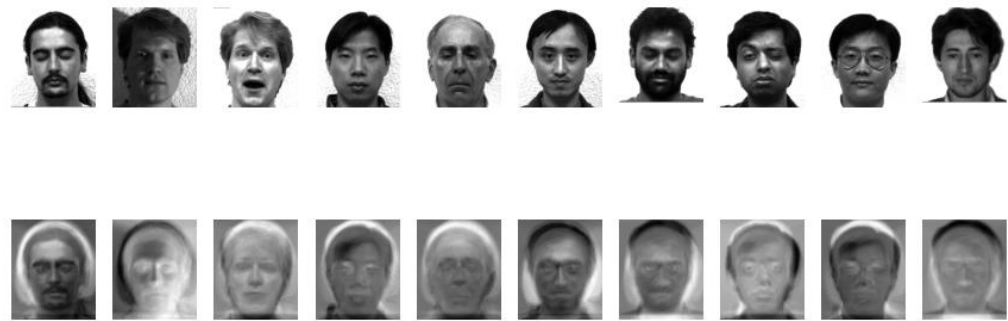
Part 1

Run the following code:

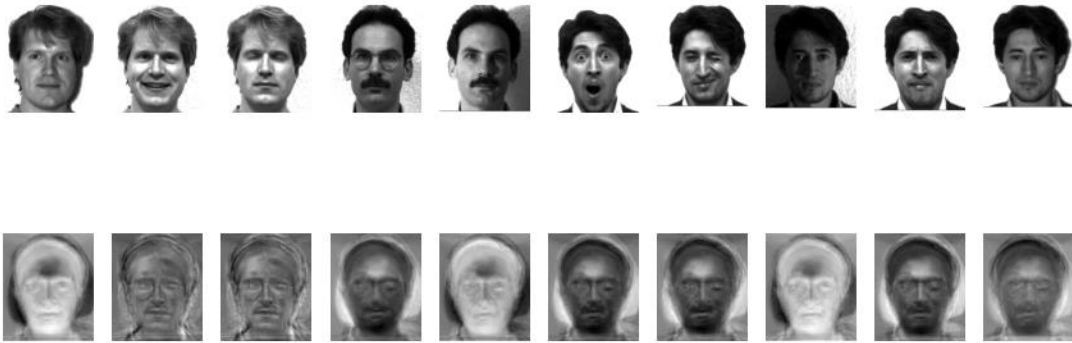
```
pca_transform, pca_z = PCA(train_image, train_label)
fisher_z, fisher_transform = LDA(pca_transform, pca_z, train_image, train_label)
```

the first 25 eigenfaces(PCA)	the first 25 fisherfaces(LDA)
	

reconstruction(PCA)



reconstruction(LDA)



Part 2

Run the following code:

```
predicted_labels = face_recognition(fisher_z, fisher_transform, train_image,  
train_label, test_image, test_label, k=3)
```

result

Accuracy: 80.00%

Part 3

Run the following code:

```
predicted_labels = face_recognition(fisher_z, fisher_transform, train_image,  
kernel = "linear"  
pca_transform, pca_z = kernel_PCA(train_image, train_label, kernel)  
fisher_z, fisher_transform = kernel_LDA(pca_transform, pca_z, train_image,  
train_label)  
predicted_labels = face_recognition(fisher_z, fisher_transform, train_image,  
train_label, test_image, test_label, kernel, k=3)  
  
kernel = "poly"  
pca_transform, pca_z = kernel_PCA(train_image, train_label, kernel)  
fisher_z, fisher_transform = kernel_LDA(pca_transform, pca_z, train_image,  
train_label)  
predicted_labels = face_recognition(fisher_z, fisher_transform, train_image,  
train_label, test_image, test_label, kernel, k=3)
```

```
kernel = "RBF"  
pca_transform, pca_z = kernel_PCA(train_image, train_label, kernel)  
fisher_z, fisher_transform = kernel_LDA(pca_transform, pca_z, train_image,  
train_label)  
predicted_labels = face_recognition(fisher_z, fisher_transform, train_image,  
train_label, test_image, test_label, kernel, k=3)
```

result

```
Kernel: linear | Accuracy: 86.67%  
Kernel: poly | Accuracy: 80.00%  
Kernel: RBF | Accuracy: 73.33%
```

In the kernel PCA with kernel LDA, the highest accuracy is achieved with the linear kernel at 86.67%, followed by the polynomial kernel at 80%, and finally, the RBF kernel at 73.33%. In contrast, when using standard PCA with LDA, the accuracy is 80%.

This suggests that the choice of kernel significantly impacts the performance of the kernel PCA method. The linear kernel, which is less complex, seems to provide the best classification performance, possibly due to the data being linearly separable or close to it. On the other hand, the polynomial and RBF kernels, which introduce non-linearity, result in slightly lower accuracy. The standard PCA with LDA achieves a solid accuracy of 80%, indicating that while kernel methods can improve performance, PCA alone can still provide a competitive result for certain datasets.

o t-SNE

Run the following code:

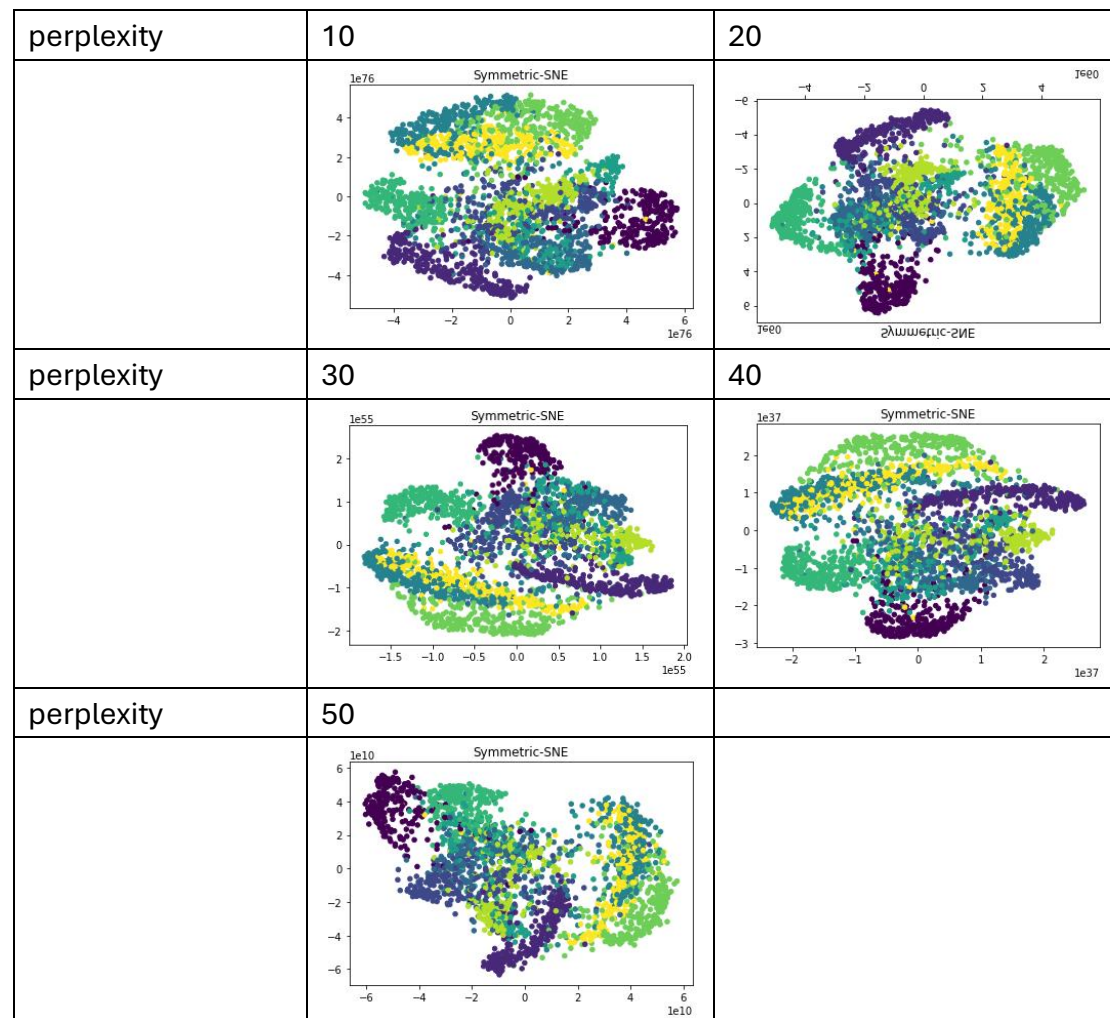
```
print("Run Y = tsne.tsne(X, no_dims, perplexity) to perform symmetric-SNE on your
dataset.")
print("Running example on 2,500 MNIST digits...")
X = np.loadtxt(r"C:\Users\yoush\Desktop\機器學習
\HW7\tsne_python\tsne_python\mnist2500_X.txt")
labels = np.loadtxt(r"C:\Users\yoush\Desktop\機器學習
\HW7\tsne_python\tsne_python\mnist2500_labels.txt")
Y, Y_history_ssne = symmetric_sne(X, 2, 50, 10.0)
save_dir = "C:\\Users\\yoush\\Desktop\\機器學習\\HW7\\ssne\\"
create_animation(Y_history_ssne, labels, "Symmetric SNE(perplexity 10)", save_dir,
"ssne_animation(perplexity_10).gif", axis_range=(-80, 80))

print("Run Y = tsne.tsne(X, no_dims, perplexity) to perform t-SNE on your
dataset.")
print("Running example on 2,500 MNIST digits...")
X = np.loadtxt(r"C:\Users\yoush\Desktop\機器學習
\HW7\tsne_python\tsne_python\mnist2500_X.txt")
labels = np.loadtxt(r"C:\Users\yoush\Desktop\機器學習
\HW7\tsne_python\tsne_python\mnist2500_labels.txt")
Y, Y_history_ssne = tsne(X, 2, 50, 10.0)
save_dir = "C:\\Users\\yoush\\Desktop\\機器學習\\HW7\\tsne\\10\\"
create_animation(Y_history_ssne, labels, "t-SNE(perplexity 10)", save_dir,
"tsne_animation(perplexity_10).gif", axis_range=(-20, 20))
```

Modifying the value of perplexity, we can get different results.

Part2

S SNE



By observing the SSNE results at varying perplexities

- Perplexity = 10 & 20

Low perplexity emphasizes local structures, leading to tightly packed clusters in the low-dimensional (low-dim) visualization.

The embeddings capture fine-grained, localized groupings, but global relationships between clusters are not well-preserved. Over-clustering or isolated group formations may occur.

- Perplexity = 30 & 40

With medium perplexities, SSNE begins to balance local and global structures.

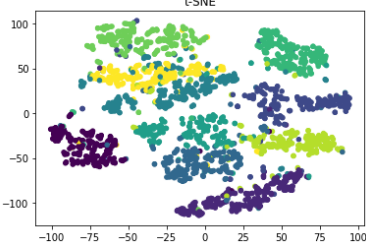
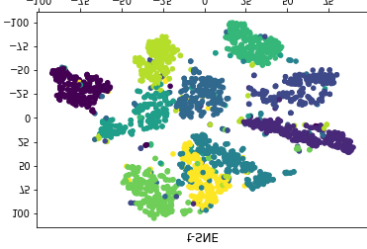
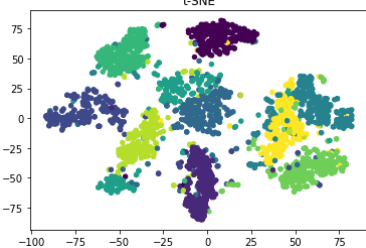
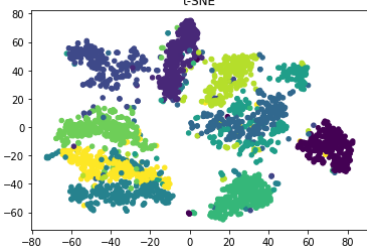
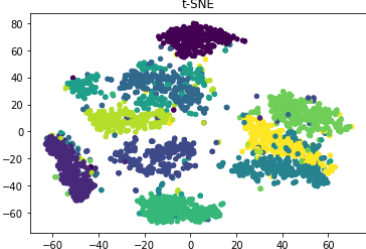
Cluster relationships become more distinct compared to lower perplexity, while intra-cluster compactness remains. The clusters are more clearly separated, though subtle group-level patterns may start to emerge.

- Perplexity = 50

High perplexity emphasizes global structures by incorporating more neighbors for each point.

This results in a smoother and less localized embedding, where broader inter-cluster relationships are captured. However, fine local details within each cluster might be sacrificed.

t SNE

perplexity	10	20
		
perplexity	30	40
		
perplexity	50	
		

Observations for t-SNE

- Perplexity = 10 & 20

Like SSNE, lower perplexities emphasize local patterns.

Clusters are compact, with well-defined local structures, but relationships between clusters (global structures) might be unclear or distorted.

Cluster overlap may be minimal, but global cohesion across the visualization may be weaker.

- Perplexity = 30 & 40

Medium perplexities balance local details and global structures, allowing for clear cluster formation and moderate separation.

Unlike SSNE, t-SNE's kernel function might exaggerate distances between clusters, making separation more apparent but potentially distorting global similarity.

- Perplexity = 50

High perplexity in t-SNE produces embeddings that prioritize global coherence. Relationships between clusters are more pronounced, with smoother transitions across the visualization.

However, local compactness of clusters may be reduced, leading to potential overlap.

Comparison between SSNE and t-SNE

- Cluster Formation

Both methods effectively group data points into clusters. However, SSNE tends to create smoother embeddings, while t-SNE might exaggerate cluster separation, leading to more visually distinct groups.

SSNE crowded problem: SSNE may suffer from the crowded problem, where clusters in the low-dimensional space are packed too closely together. This occurs when the embedding overly compresses global distances, making it harder to visually interpret inter-cluster relationships compared to t-SNE.

- Local vs. Global Structure

SSNE generally achieves a better balance between local and global structures across perplexities, but the crowded problem can limit its ability to preserve meaningful inter-cluster distances.

t-SNE, while also struggling with global structures, tends to overemphasize local compactness, which can distort global relationships between clusters.

- Perplexity Sensitivity

t-SNE is highly sensitive to perplexity changes, with noticeable shifts in visualization quality as perplexity increases or decreases.

SSNE appears more robust across different perplexities but is more prone to compressing clusters together due to its focus on preserving smooth global distributions.

- Interpretability

SSNE embeddings are often more interpretable in terms of global structure but may suffer from overlaps or cluster crowding.

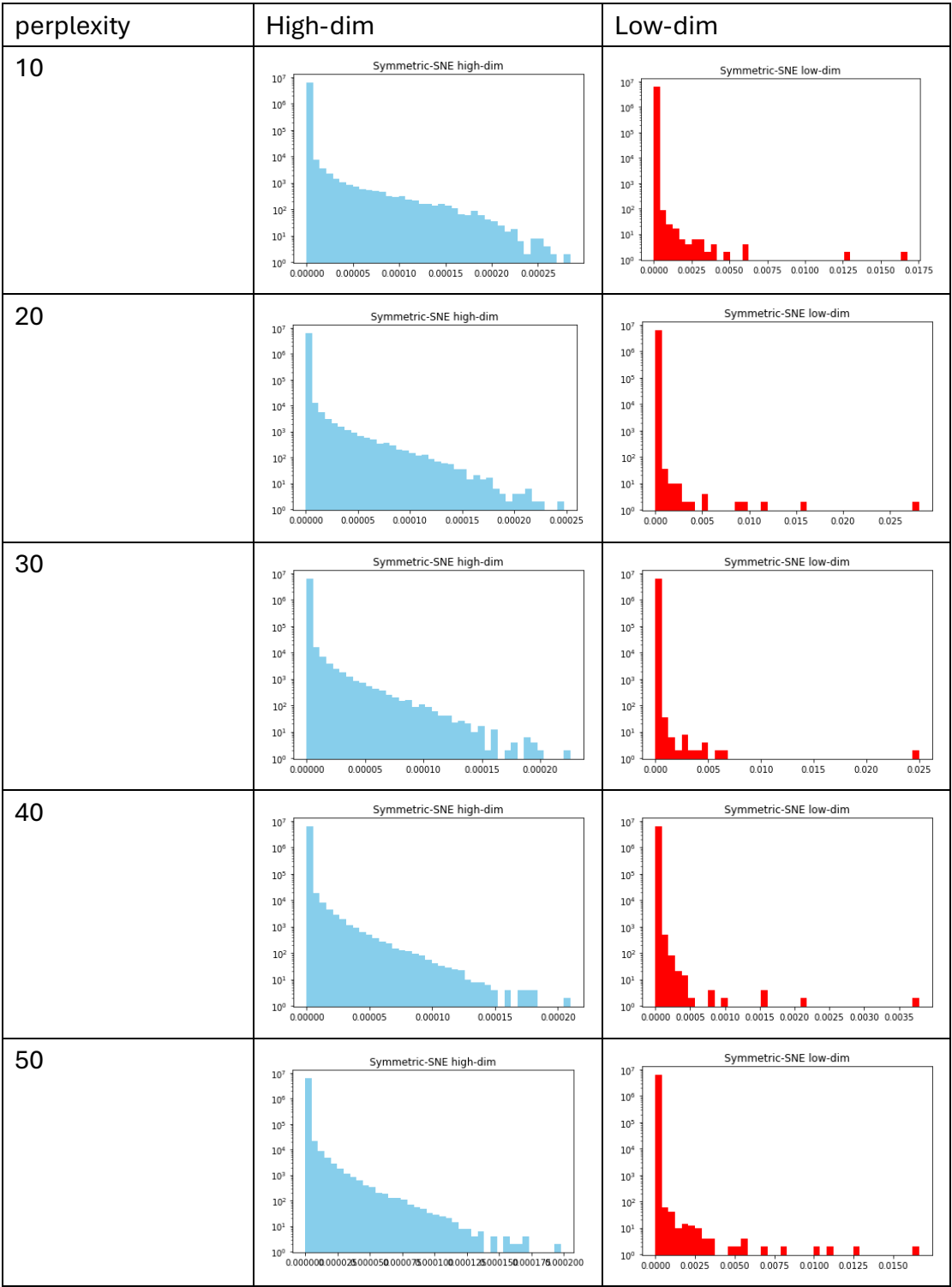
t-SNE emphasizes distinct visual clustering, which might make it more appealing for exploratory data analysis but can misrepresent global relationships.

- Summary

By including the crowded problem in SSNE, this section emphasizes one of its limitations in balancing global and local structures, making it clear that while SSNE provides smoother embeddings, t-SNE can sometimes offer more interpretable visual clusters despite its own limitations. Additionally, with smaller perplexity values, both SSNE and t-SNE tend to achieve clearer separations between clusters compared to larger perplexity values.

similarities

S SNE



SSNE Similarities Analysis

- Perplexity = 10 & 20:

At low perplexities, high-dimensional similarities are sharply concentrated around local neighborhoods (a peaked distribution).

SSNE performs well at preserving these local structures, as the low-dim similarities closely follow the high-dim patterns. However, global relationships between clusters may not be adequately represented.

- Perplexity = 30 & 40:

With medium perplexities, the high-dimensional similarity distribution becomes smoother, as more neighbors are included in the calculations.

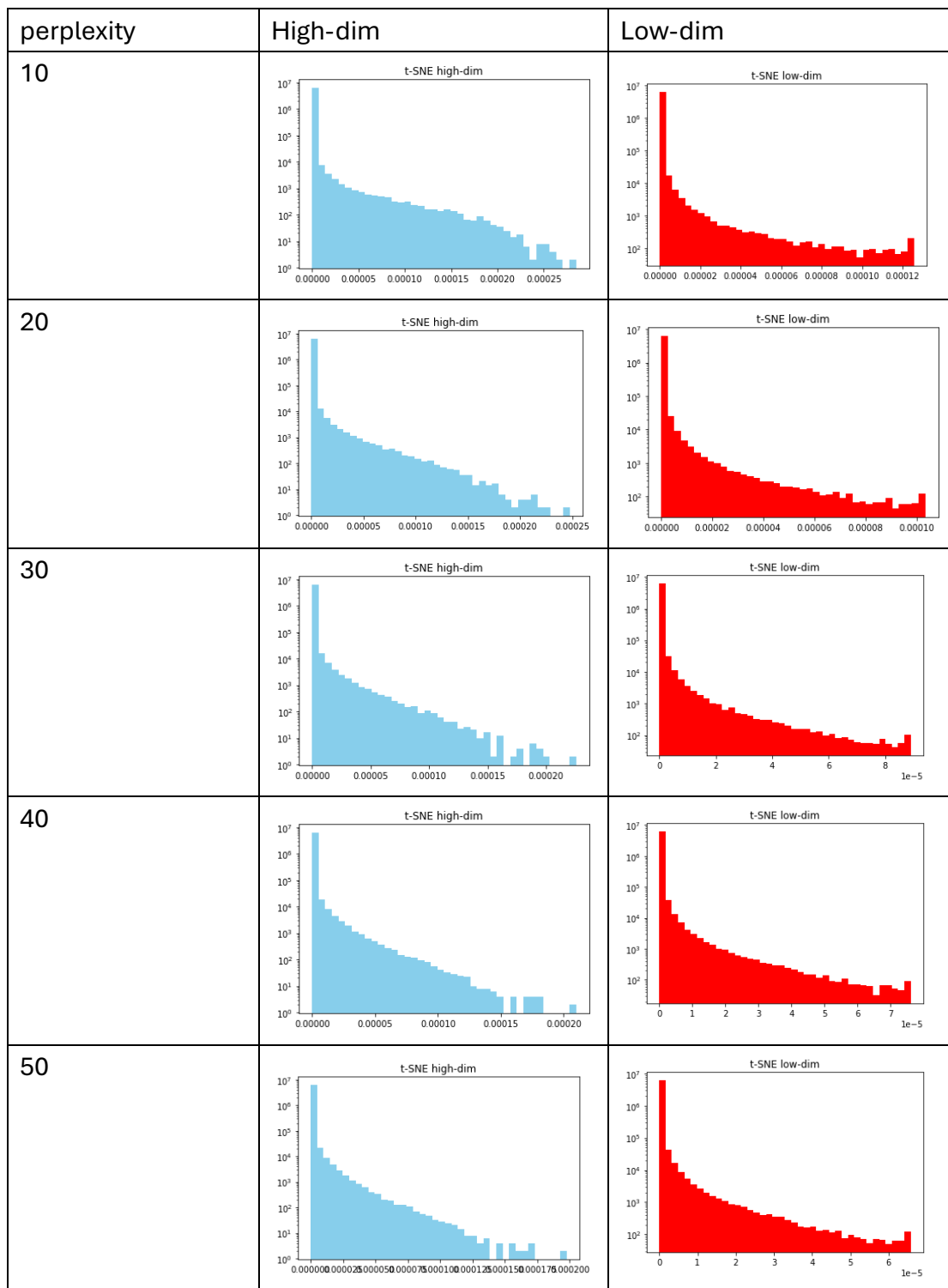
SSNE balances local and global structures, and the low-dim similarity distribution begins to align more consistently with the high-dim similarity distribution.

- Perplexity = 50:

At higher perplexities, the high-dim similarity distribution is significantly smoothed, reflecting a focus on global relationships.

SSNE captures these global patterns effectively, with the low-dim similarity distribution maintaining a close match to the high-dim similarity distribution. However, some fine local details may be sacrificed.

t SNE



t-SNE Similarities Analysis

- Perplexity = 10 & 20:

Low perplexity heavily emphasizes local structures, resulting in high-dim

similarities that are sharply peaked around close neighbors.

t-SNE tends to exaggerate local clusters in the low-dim space, leading to tightly packed clusters. However, this can distort the global structure by over-separating clusters.

- Perplexity = 30 & 40:

Medium perplexities introduce a balance between local and global structures in the high-dim similarity distribution.

t-SNE captures local clusters well, but its kernel often pushes clusters further apart, creating more exaggerated low-dim similarities compared to high-dim similarities.

- Perplexity = 50:

High perplexities smooth the high-dim similarity distribution, capturing global patterns.

In the low-dim space, t-SNE still emphasizes separation between clusters, which may lead to visual artifacts such as overly separated or distorted cluster relationships.

Comparison of SSNE and t-SNE Similarities

- Preservation of Local and Global Structures

SSNE provides a smoother and more balanced transition from high-dim to low-dim similarities, maintaining both local and global relationships.

t-SNE emphasizes local clustering but may distort global relationships, especially at low and high perplexities.

- Effect of Perplexity

SSNE exhibits consistent performance across varying perplexities, with low-dim similarities closely following high-dim distributions.

t-SNE is more sensitive to perplexity, with pronounced differences in how

similarities are preserved at different levels.

- Cluster Interpretability

t-SNE often creates more visually distinct clusters, which may not faithfully reflect the high-dim similarities.

SSNE provides embeddings that are more representative of the original data distribution, though the clusters may appear less visually separated.

3. Observations and Discussion

I. Meaning of Eigenfaces

- **Definition:** Eigenfaces are the principal components derived from PCA, representing the directions of maximum variance in the dataset. These are essentially a set of basis images that, when linearly combined, can approximate any image in the dataset.
- **Intuition:** The first few eigenfaces often capture large-scale variations such as lighting conditions, general facial structure, and orientation. Subsequent eigenfaces add finer details but contribute less to the overall variance.
- **Usefulness:** Eigenfaces allow for dimensionality reduction by projecting high-dimensional face images onto a lower-dimensional subspace while retaining as much variance as possible.

II. Comparison of Dimensionality Reduction Methods

- **PCA (Principal Component Analysis)**
 - **Advantages:**
 - Efficient for dimensionality reduction and data compression.
 - Captures global patterns and variations in the dataset.
 - Unsupervised, making it easy to apply to any dataset.
 - **Disadvantages:**
 - Focuses on variance rather than class separability, making it less effective for classification tasks.
 - Sensitive to noise and irrelevant features in the data.
- **LDA (Linear Discriminant Analysis)**
 - **Advantages:**
 - Maximizes class separability, making it well-suited for

classification tasks such as face recognition.

- Produces Fisherfaces, which emphasize discriminative features between classes.
- Disadvantages:
 - Assumes linear separability of data, limiting its performance on non-linear datasets.
 - Requires labeled data, making it less flexible than PCA.
- Kernel PCA
 - Advantages:
 - Extends PCA to capture non-linear relationships in the data using kernel functions (e.g., polynomial, RBF).
 - More flexible in handling complex datasets compared to standard PCA.
 - Disadvantages:
 - Choice of kernel and hyperparameters (e.g., degree in polynomial kernels) significantly affects performance.
 - Computationally expensive for large datasets.
- Kernel LDA
 - Advantages:
 - Combines the strengths of LDA and kernel methods to achieve non-linear class separability.
 - Suitable for datasets with complex boundaries between classes.
 - Disadvantages:
 - Computationally intensive and sensitive to kernel choice.
 - Requires labeled data and careful tuning.

III. Observations and Comparisons

- Reconstruction Quality:

PCA reconstruction focuses on global features, producing more generalized approximations of faces.

LDA reconstruction, though less detailed, emphasizes class-specific features, making it better for distinguishing between individuals.

- Face Recognition Accuracy:

LDA and Kernel LDA outperform PCA and Kernel PCA for face recognition due to their focus on class separability.

Kernel methods (e.g., polynomial kernel) generally improve performance by capturing non-linear patterns, especially in datasets with complex distributions.

- Trade-offs:

PCA is faster and simpler but less effective for classification.

LDA and Kernel methods require more computation and parameter tuning but offer better results for tasks like face recognition.

IV. Suggestions for Future Exploration

Experiment with different kernels (e.g., RBF, sigmoid) for Kernel PCA and Kernel LDA to see their effects on reconstruction and recognition accuracy.

Compare the results of PCA, LDA, and their kernel versions on datasets with varying complexity (e.g., linear vs. non-linear separability).

Test hybrid approaches, such as combining PCA for initial dimensionality reduction followed by LDA for classification.