

Homework 7 Report

312657018 統計碩二 陳佳媛

1. Code with detailed explanations

A. Kernel Eigenfaces

Part 1.

Part 2.

Part 3.

B. t-SNE

Part 1.

Part 2.

Part 3.

Part 4.

2. Experiments and Discussion

A. Kernel Eigenfaces

a. PCA

b. LDA

B. t-SNE

a. Embedding

b. Pairwise Similarities

3. Observations and Discussion

A. Kernel Eigenfaces

B. t-SNE

1. Code with detailed explanations

A. Kernel Eigenfaces

Part 1.

Utils

```

class Utils:
    @staticmethod
    def eigen_decomposition(S):
        e_values, e_vectors = np.linalg.eig(S)
        idx = np.argsort(e_values)[::-1][:25]
        e_values = e_values[idx]
        e_vectors = e_vectors[:, idx]

        pos_idx = np.where(e_values > 0)[0]
        e_values = e_values[pos_idx].real
        e_vectors = e_vectors[:, pos_idx].real

        return e_values, e_vectors

    @staticmethod
    def kernel(x, kernel_f=None):
        if kernel_f == None:
            return np.cov(x, bias=True)

        elif kernel_f == 'linear':
            K = x @ x.T
        elif kernel_f == 'poly':
            K = (x @ x.T + 1e-5)**3
        elif kernel_f == 'RBF':
            K = np.exp(-1e-8*cdist(x, x, 'sqeuclidean'))

        N = x.shape[0]
        one_N = np.ones((N, N)) / N
        Kc = K - one_N@K - K@one_N + one_N@K@one_N
        return Kc

```

The `Utils` class provides some functions for implementing PCA, LDA, and kernel functions(show in part3).

1. `eigen_decomposition(S)` : Performs eigenvalue decomposition on the input matrix `S`, extracting the top 25 largest eigenvalues and their corresponding eigenvectors. Only the real parts of these eigenvalues and eigenvectors are retained and returned.
2. `kernel(x, kernel_f=None)` : computes the kernel matrix for the given input data `x` based on the specified kernel function `kernel_f`.
 - `x` : The input data matrix, where rows represent samples, and columns represent features.
 - `kernel_f` : The type of kernel function to use (`None` , `linear` , `poly` , or `RBF`).
 - `None` : return the standard covariance matrix of `x`, which is used in simple PCA.
 - `linear` , `poly` and `RBF` will be explained and used in part3.

PCA & LDA

```

class Embedding(Utils):
    def __init__(self, X, y, kernel_f=None):
        self.X = X
        self.y = y
        self.kernel_f = kernel_f
        self.X_mean = np.mean(self.X, axis=0)
        self.X_center = self.X - self.X_mean

    def PCA(self):
        S = self.kernel(self.X.T, self.kernel_f) #(696, 696)

        _, e_vectors = self.eigen_decomposition(S)
        transform = e_vectors # (696, 25)
        W = self.X_center @ e_vectors # (135, 25)
        z = W.T @ self.X_center # (25, 696)
        reconstruct = W @ z + self.X_mean.reshape(1, -1) # (135, 696)

        return transform, W, z, reconstruct

```

The `Embedding` class extends the `Utils` class and provides methods for implementing LDA and LDA, both critical techniques in dimensionality reduction.

1. `__init__`

- `X`: The input data matrix of shape $n \times d$ (n: samples, d: features).
- `y`: Class labels for the samples.
- `kernel_f`: The kernel function to use.
- `X_mean`: mean of across each feature.
- `X_center`: Center X by subtracting the mean: .

2. `PCA` step:

- a. `self.kernel` computes the covariance matrix S of X^T , where X is shaped as (samples,features).

X^T is used because the goal is to reduce the dimensionality of the features and extract the most important components. The covariance matrix S captures the relationships between pixels. Additionally, resizing the original images from 231×192 to 29×24 reduces computational cost while retaining sufficient structural information.

- b. `self.eigen_decomposition` solve the eigenvalue problem for S.
- c. Compute projections W by multiplying the centered data with the eigenvectors:

$$W = X_{\text{centered}} V$$

where V is the matrix of top eigenvectors.

- d. Reconstruct the original data from the projections:

$$\hat{X} = WW^T + \mu$$

where μ is the mean vector.

```
def LDA(self, pca_z):
    mu = np.mean(pca_z, axis=1)
    N = pca_z.shape[1]
    Sw, Sb = np.zeros((N, N)), np.zeros((N, N))

    labels = np.unique(self.y)
    for i in range(len(labels)):
        z_j = self.X[self.y == i + 1]
        mu_j = np.mean(z_j, axis=0, keepdims=True).T

        Sw_j = np.cov(z_j.T, bias=True)
        Sw += Sw_j

        n_j = z_j.shape[1]
        Sb += n_j * (mu_j - mu) @ (mu_j - mu).T

    S = np.linalg.pinv(Sw) @ Sb # (696, 696)
    _, e_vectors = self.eigen_decomposition(S)
    transform = e_vectors # (696, 25)

    W = self.X_center @ e_vectors # (135, 25)
    z = W.T @ self.X_center # (25, 696)
    reconstruct = W @ z + self.X_mean.reshape(1, -1)

    return transform, W, z, reconstruct
```

3. **LDA** step:

- a. Compute the mean of each class:

$$\mu_c = \frac{1}{n_c} \sum_{i \in c} X_i$$

where c is the class index and n_c is the number of samples in class c .

pca_z represents the data projected into the PCA space. Using **pca_z** as input for LDA serves as a preprocessing step to enhance LDA's effectiveness by reducing both dimensionality and noise in the data. This ensures cleaner and more meaningful computation of the within-class and between-class scatter matrices, ultimately improving the overall performance of the LDA method.

- b. Compute Within-Class Scatter (S_w), measuring the spread of samples within each class.:

$$S_w = \sum_c \sum_{i \in c} (X_i - \mu_c)(X_i - \mu_c)^T$$

- c. Between-Class Scatter (S_b), measuring the separation between class means.:

$$S_b = \sum_c n_c (\mu_c - \mu)(\mu_c - \mu)^T$$

- d. Use `self.eigen_decomposition` to solve the eigenvalue problem:

$$S_w^{-1} S_b v = \lambda v$$

where v is Eigenvectors and λ is Eigenvalues.

- e. The subsequent steps same as steps (c) to (d) in PCA.

Visualization Functions

```
def plot_w(W, method, kernel_f, n_components=25):
    plt.figure(figsize=(6, 8))
    for i in range(n_components):
        plt.subplot(5, 5, i+1)
        plt.axis('off')
        plt.imshow(W[:, i].reshape(29, 24), cmap="gray")
    # plt.show()
    plt.savefig(f"./HW7/Eigenfaces/transform_{method}_{kernel_f}.jpg", format='JPEG')
    plt.close()

def plot_reconstruct(re_img, ori_img, method, kernel_f):
    plt.figure(figsize=(6, 8))
    for i in range(10):
        plt.subplot(5, 4, i*2+1)
        plt.axis('off')
        plt.imshow(re_img[i, :].reshape(29, 24), cmap='gray')
        plt.subplot(5, 4, i*2+2)
        plt.axis('off')
        plt.imshow(ori_img[i].reshape(-1, 1).reshape(29, 24), cmap='gray')
    # plt.show()
    plt.savefig(f"./HW7/Eigenfaces/reconstruct_{method}_{kernel_f}.jpg", format='JPEG')
    plt.close()
```

1. `plot_w`: Visualizes the top `n_components` eigenfaces or fisherfaces.
2. `plot_reconstruct`: print reconstructed images and compare with original ones.

Part 2.

```
def test(self, X_test, y_test, method, k=15):
    test_center = X_test - self.X_mean

    if method == 'PCA':
        transform, W, z, _ = self.PCA()
    elif method == 'LDA':
        pca_transform, pca_W, pca_z, _ = self.PCA()
        transform, W, z, _ = self.LDA(pca_W, pca_z)

    W_test = test_center @ transform # (30, 25)
    # print(z_test.shape, z.shape)
    # z_test = W.T @ test_center.T @ test_center # (25, 696)
    dist_matrix = cdist(W_test, W, metric='sqeuclidean') # (30, 135)

    preds = []
    for i in range(dist_matrix.shape[0]):
        k_idx = np.argsort(dist_matrix[i])[:k]

        k_labels = self.y[k_idx]
        pred = np.argmax(np.bincount(k_labels))
        preds.append(pred)

    preds = np.array(preds)
    gts = np.sum(preds == y_test)
    acc = gts / X_test.shape[0]
    loss = (X_test.shape[0] - gts) / X_test.shape[0]
    no_gts = X_test.shape[0] - gts

    print(f'==== {method} {self.kernel_f} Kernel =====')
    print(f'Accuracy: {acc:.2%}')
    print(f'Correct samples: {gts}')
    print(f'Loss: {loss:.2%}')
    print(f'Wrong samples: {no_gts}')
```

`test` evaluates the performance of PCA or LDA in a classification task by using the k-NN algorithm.

1. Input:

- a. `X_test`: Test data matrix ($m \times d$).
- b. `y_test`: Labels for the test data.
- c. `method`: Specifies whether to use PCA or LDA for dimensionality reduction.
- d. `k`: Number of neighbors for the k-NN algorithm (default is 15).

2. Centers the test data by subtracting the mean of the training data (`X_mean`) to ensure consistency in preprocessing. Applies the specified method (PCA or LDA) to perform

dimensionality reduction, resulting in the transform matrix, W , and z . Subsequently, projects the centered test data into the same reduced-dimensional space as the training data using the transform matrix, calculated as

$$W_{\text{test}} = \text{test_center} @ \text{transform}$$

3. k-NN is highlighted in the red box, for each test sample:
 - Identify the indices of the k-nearest training samples based on their distances.
 - Retrieve the labels of these k nearest neighbors.
 - Use majority voting (`argmax(np.bincount)`) to predict the test sample's label.
4. Compute accuracy and loss then print it.

Part 3.

The process for PCA, LDA, including visualization and testing outcomes, are consistent with the earlier parts, only the kernel function is modified in this part.

```
@staticmethod
def kernel(x, kernel_f=None):
    if kernel_f == None:
        return np.cov(x, bias=True)

    elif kernel_f == 'linear':
        K = x @ x.T
    elif kernel_f == 'poly':
        K = (x @ x.T + 1e-5)**3
    elif kernel_f == 'RBF':
        K = np.exp(-1e-8*cdist(x, x, 'sqeuclidean'))

    N = x.shape[0]
    one_N = np.ones((N, N)) / N
    Kc = K - one_N@K - K@one_N + one_N@K@one_N
    return Kc
```

kernel is defined in part1, in this homework, I use 3 types of kernel function:

- `linear`: Computes $K = xx^T$, which is the linear kernel matrix.
- `poly`: Computes $K = (xx^T + c)^d$, here $c = 1e^{-5}$ and $d = 3$, which is a polynomial kernel.
- `RBF`: Computes $K = \exp(-\gamma \cdot ||x_i - x_j||^2)$, here $\gamma = 1e^{-8}$, which is a RBF kernel.

Performs double centering of the kernel matrix K to ensure it is centered in feature space:

$$K_c = K - \frac{1}{N} \mathbf{1}_N K - K \frac{1}{N} \mathbf{1}_N + \frac{1}{N^2} \mathbf{1}_N K \mathbf{1}_N$$

where $\mathbf{1}_N$ is an $N \times N$ matrix of ones, and N is the number of samples.

B. t-SNE

Part 1.

```
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))
```

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

a. Formula for t-SNE

t-SNE minimizes the KL divergence between high-dimensional similarities P and low-dimensional similarities Q , weighted by the t-distribution:

- High-Dimensional Similarities P_{ij} :

D

- P_{ij} represents the probability that i picks j as a neighbor.
- The bandwidth σ_i is adjusted per data point using perplexity.
- Low-Dimensional Similarities Q_{ij} :

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

- Q_{ij} models similarities in the low-dimensional space.
- The t-distribution's heavy tails alleviate the crowding problem.
- Cost Function:

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

- Gradient:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

b. Formula for Symmetric SNE

Symmetric SNE also minimizes the KL divergence but uses a Gaussian kernel for Q_{ij} , similar to P_{ij} :

- High-Dimensional Similarities P_{ij} same as in t-SNE.
- Low-Dimensional Similarities Q_{ij} :

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

- Cost Function:

$$C = KL(P||Q) = \sum_i \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

- Gradient:

$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

Part 2.

```
def save_embedding_with_gif(Y, labels, mode, perplexity, iter_num, image_files):
    """
    Saves the current embedding as a frame and appends it for GIF creation.
    """
    plt.figure(figsize=(8, 8))
    for label in np.unique(labels):
        idx = labels == label
        plt.scatter(Y[idx, 0], Y[idx, 1], label=f'Class {label}', alpha=0.7, s=10)
    # plt.legend()
    plt.title(f"Iteration {iter_num}: {'t-SNE' if mode == 0 else 'Symmetric SNE'}_Perplexity={perplexity}")
    plt.axis('off')
    filename = f"{'t-SNE' if mode == 0 else 'Symmetric SNE'}_Perplexity{perplexity}_{iter_num:04d}.png"
    plt.savefig(filename)
    plt.close()
    image_files.append(filename)

def create_gif_and_save_final(image_files, output_gif, final_output, duration=100):
    """
    Creates a GIF from captured frames and saves the final embedding as a static image.
    """
    # Create GIF from all frames
    frames = [Image.open(img) for img in image_files]
    frames[0].save(output_gif, save_all=True, append_images=frames[1:], duration=duration, loop=0)
    print(f"GIF saved as {output_gif}")

    # Save the last frame as the final embedding
    if image_files:
        os.rename(image_files[-1], final_output)
        print(f"Final embedding saved as {final_output}")

    # Clean up intermediate frame images
    for img in image_files[:-1]:
        os.remove(img)
```

```
# Compute current value of cost function
if (iter + 1) % 50 == 0:
    C = np.sum(P * np.log(P / Q))
    print("Iteration %d: error is %f" % (iter + 1, C))
    save_embedding_with_gif(Y, labels, mode, perplexity, iter + 1, img)
    if prev_cost is not None and abs(prev_cost - C) < threshold:
        print(f"Converged at iteration {iter + 1} with error {C:.6f}")
        break
    prev_cost = C

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

plot_similarity(P, Q, mode, perplexity)
return Y, img
```

1. `save_embedding_with_gif`: To generate and save a visualization of the embedding at a specific iteration and append the filename to a list for GIF creation.

- input
 - `Y`: The current embedding ($n \times 2$), where n is the number of data points.
 - `labels`: Class labels for the data points.

- `mode` : t-SNE (0) or Symmetric SNE (1).
 - `perplexity`
 - `iter_num` : The current iteration number.
 - `image_files` : A list to store filenames of saved frames.
 - Purpose:
 - Creates a scatter plot of the embedding, each class is plotted with a different color.
 - `Y[idx,0]` and `Y[idx, 0]`: X and Y coordinates of the embedding for class label.
2. `create_gif_and_save_final` : To combine the saved embedding images into a GIF, save the final embedding as a static image, and clean up intermediate images.
- Input:
 - `image_files`
 - `output_gif` : The filename for the resulting GIF.
 - `final_output` : The filename for the final embedding image.
 - `duration` : Duration for each frame in the GIF.
 - Purpose:
 - Create GIF visualizing the embedding over iterations
 - Renames the last frame (final embedding visualization) to `final_output`.
 - Deletes all frames except the last one to free up storage.

Part 3.

```
def plot_similarity(P, Q, mode=0):
    """
    Plots the distribution of pairwise similarities for high-dimensional and low-dimensional spaces.

    Parameters:
    - P: Pairwise similarities in high-dimensional space.
    - Q: Pairwise similarities in low-dimensional space.
    - mode: 0 for t-SNE, 1 for symmetric SNE.
    """
    plt.subplot(2, 1, 1)
    plt.title('Symmetric SNE high-dim' if mode else 't-SNE high-dim')
    plt.hist(P.flatten(), bins=40, log=True)

    plt.subplot(2, 1, 2)
    plt.title('Symmetric SNE low-dim' if mode else 't-SNE low-dim')
    plt.hist(Q.flatten(), bins=40, log=True)

    plt.tight_layout()
    plt.savefig('SSNE.jpg' if mode else 't-SNE.jpg')
    plt.close()
```

`plot_similarity` generates histograms of pairwise similarities for both high-dimensional and low-dimensional spaces, helping to analyze the embedding quality and compare methods (t-SNE vs symmetric SNE). It highlights how the embedding preserves relationships and adapts to different similarity measures.

Part 4.

```
perplexitys = [10, 20, 30, 40, 50] # 不同的 perplexity 值
modes = [0, 1] # 0: t-SNE, 1: Symmetric SNE

for pp in perplexitys:
    for m in modes:
        print(f'=== Start {"t-SNE" if not m else "Symmetric SNE"} with perplexity={pp} ===')
        try:
            y, img = tsne(x, no_dims=2, initial_dims=50, perplexity=pp, mode=m)

            output_gif = f'{"t-SNE" if not m else "symmetric SNE"}_{pp}.gif'
            final_output = f'{"t-SNE" if not m else "symmetric SNE"}_{pp}.png'
            create_gif_and_save_final(img, output_gif=output_gif, final_output=final_output)

            print(f"Saved GIF: {output_gif} and final image: {final_output}")
        except ValueError as e:
            print(f"Error during SNE with mode={m}, perplexity={pp}: {e}")
```

Use for loop to run **t-SNE** and **symmetric SNE** for a range of perplexity values, generates visualizations at each step, and saves the results as GIFs and final images.

2. Experiments and Discussion

A. Kernel Eigenfaces


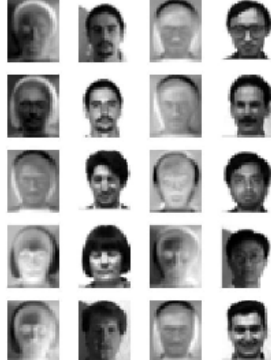



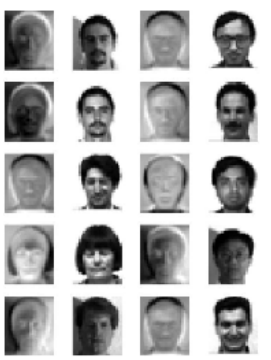

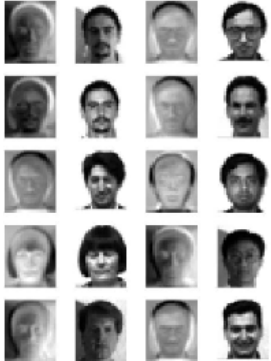
```
kernel_f = [None, 'linear', 'poly', 'RBF']
for f in kernel_f:
    embedding = Embedding(X_train, y_train, kernel_f=f)
    PCA_transform, PCA_W, PCA_z, PCA_re = embedding.PCA()
    embedding.test(X_test, y_test, method='PCA')
    plot_w(PCA_transform, kernel_f=f, method='PCA')
    plot_reconstruct(PCA_re, X_train, kernel_f=f, method='PCA')

    LDA_transform, LDA_W, LDA_x, LDA_re = embedding.LDA(PCA_W, PCA_z)
    embedding.test(X_test, y_test, method='LDA')
    plot_w(LDA_transform, kernel_f=f, method='LDA')
    plot_reconstruct(LDA_re, X_train, kernel_f=f, method='LDA')
```

The loop repeats for each kernel function, allowing:

- Visualization of how different kernels affect the eigenfaces (PCA) and fisherfaces (LDA).
- Testing to evaluate how each kernel function impacts classification accuracy.

a. PCA

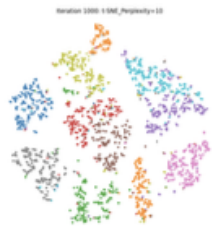
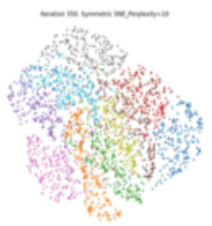


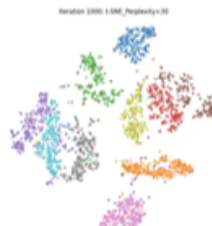




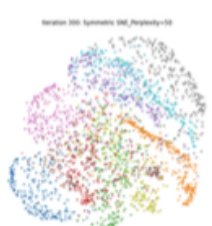
	Eigenfaces	Reconstructed	Accuracy
Simple			80.00%
Linear kernel			86.67%
Polynomial kernel			76.67%
RBF kernel			86.67%

b. LDA

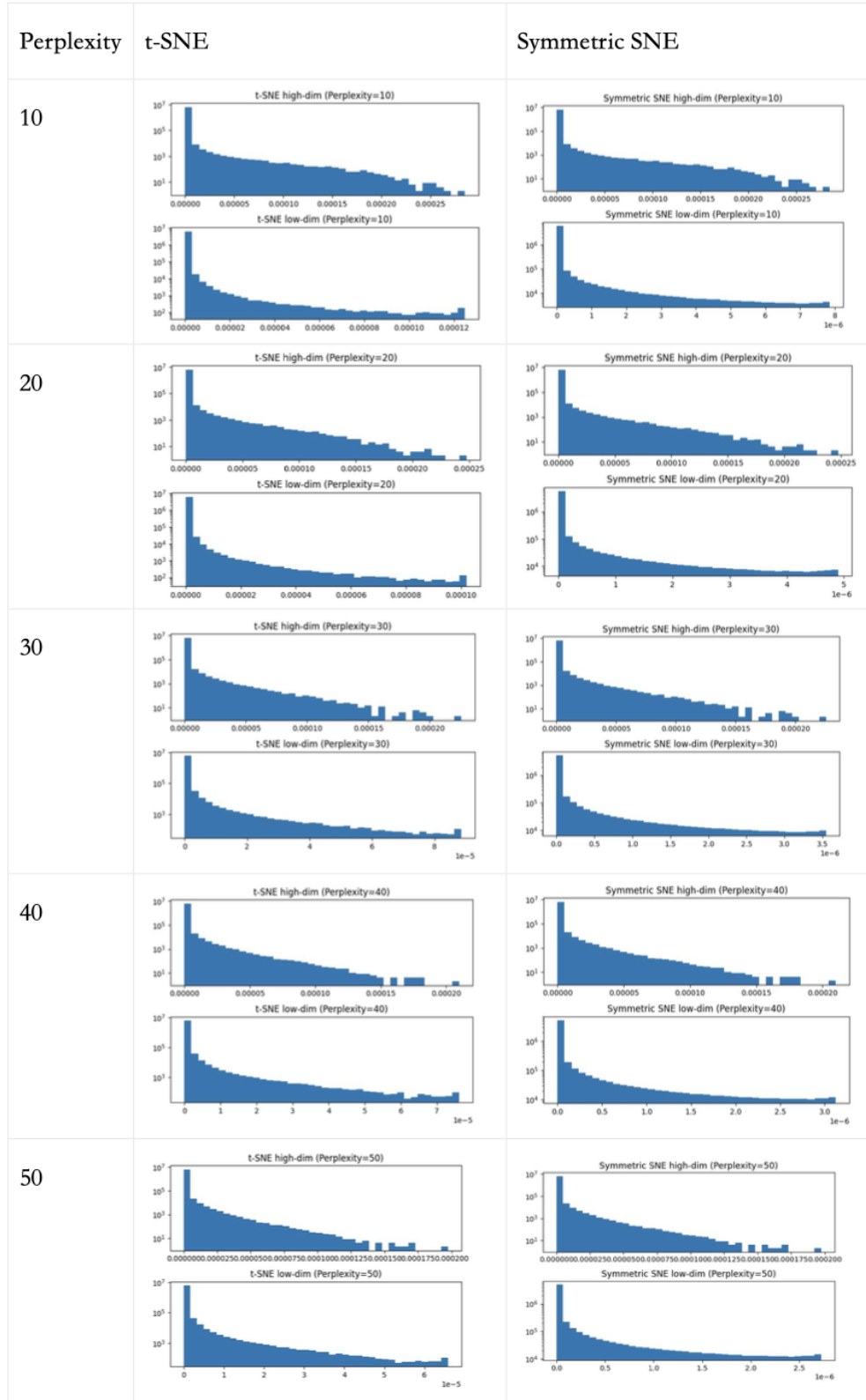
	Eigenfaces	Reconstructed	Accuracy
Simple			96.67%
Linear kernel			96.67%
Polynomial kernel			96.67%
RBF kernel			96.67%

B. t-SNE

a. Embedding

Perplexity	t-SNE	Symmetric SNE
10		
20		
30		
40		
50		

b. Pairwise Similarities



3. Observations and Discussion

A. Kernel Eigenfaces

- a. The reconstructed faces from PCA are clearer than those from LDA.
- b. The performance of LDA is better than that of PCA. However, it is worth noting that the performance of the kernel version highly depends on the choice of hyperparameters. Initially, when the hyperparameters were set to larger values, the accuracy was poor, with LDA even performing worse than PCA. After tuning the hyperparameters to smaller values, the accuracy improved significantly.
- c. Eigenfaces depict the contour of the faces, thereby extracting the features of the images. Based on these features, we can effectively classify the testing images.

B. t-SNE

- a. From the scatter plots, symmetric SNE suffers from the crowding problem. Without color, it is difficult to distinguish between different classes. In contrast, t-SNE uses the t-distribution to alleviate the crowding problem, maintaining larger distances for points that are far apart in high-dimensional space.
- b. Symmetric SNE converges faster than t-SNE, as observed from the results in the GIF files.
- c. Perplexity affects the number of neighbors considered. A higher perplexity reduces sensitivity to smaller clusters. Due to the crowding problem in symmetric SNE, the effect of perplexity is not significant. However, in t-SNE, the effect becomes more noticeable as perplexity increases.
- d. Low vs. High Perplexity:
 - Low Perplexity focuses on preserving local structures, resulting in clearer representations of smaller clusters but possibly sacrificing global structure.
 - High Perplexity Balances local and global structures better, producing more cohesive embeddings. However, excessive perplexity may overemphasize global structure, compressing smaller clusters.