ML Homework 6 Report

陳宥銓
313554013
Institute of Data Science and Engineering

Contents

1 Code

1.1 Kernel Eigenface

1.1.1 Data Loader and Main Function

Fig.1 shows the function the function used for loading data. For an input image with size (w, h), we first resize it to$(w' = \frac{w}{c}, h' = \frac{h}{c})$, where $c \geq 0$. The resized image is then reshaped into $(1, w'h')$ and all data are stacked into an $(n, w'h')$ array.

```python
def dataLoader(root, resize_factor=0):
    img_data = []
    label = []
    for file in sorted(os.listdir(root)):
        img_path = os.path.join(root, file)
        img = Image.open(img_path)

        w, h = img.size
        if resize_factor > 0: # Resize
            w, h = w//resize_factor, h//resize_factor
            img = img.resize((w, h))

        img = np.array(img).reshape(-1) # Reshape to 1*n, n=w*h
        img_data.append(img)
        label.append(int(file[7:9]))
    return np.array(img_data, dtype=np.float64), np.array(label), w, h
```

Figure 1: Data Loader

```python
def main(opt):
    imageTr, labelTr, w_Tr, h_Tr = dataLoader(opt["dataTr_path"], resize_factor=opt["resize"])
    imageTs, labelTs, _, _ = dataLoader(opt["dataTs_path"], resize_factor=opt["resize"])

    if opt["task"] == "PCA":
        W = pca(imageTr, opt)
        plot_eigenface(W, w_Tr, h_Tr, opt)
        reconstruction(W, imageTr, w_Tr, h_Tr, opt)
        recognition(imageTr, labelTr, imageTs, labelTs, W, opt)

    elif opt["task"] == "LDA":
        W = lda(imageTr, labelTr, opt)
        plot_eigenface(W, w_Tr, h_Tr, opt)
        reconstruction(W, imageTr, w_Tr, h_Tr, opt)
        recognition(imageTr, labelTr, imageTs, labelTs, W, opt)

    elif opt["task"] == "kernelPCA":
        kernelTr = build_kernel(imageTr, imageTr, opt)
        kernelTs = build_kernel(imageTs, imageTr, opt)
        W = pca(kernelTr, opt, kernel=True)
        recognition(kernelTr, labelTr, kernelTs, labelTs, W, opt)

    elif opt["task"] == "kernelLDA":
        kernelTr = build_kernel(imageTr, imageTr, opt)
        kernelTs = build_kernel(imageTs, imageTr, opt)
        W = lda(kernelTr, labelTr, opt)
        recognition(kernelTr, labelTr, kernelTs, labelTs, W, opt)
```

Figure 2: Main Function

1.1.2 Kernel

To perform kernel PCA and kernel LDA, we construct the following three types of kernel

1. Linear kernel
2. RBF kernel
3. Linear RBF kernel

Once the kernel K is constructed, we centralize it

$$
\begin{aligned}
K^c =& (\phi(X) - mean(\phi(X)))(\phi(X) - mean(\phi(X)))^T && (1)\\
=& \phi(X)\phi(X)^T - mean(\phi(X))\phi(X) && (2)\\
& - \phi(X)mean(\phi(X)) + mean(\phi(X))mean(\phi(X)) && (3)\\
=& K - mean_{row}(K) - mean_{col}(K) - mean(K) && (4)
\end{aligned}
$$

```python
def center_kernel(K):
    row_mean = np.mean(K, axis=1, keepdims=True)
    col_mean = np.mean(K, axis=0, keepdims=True)
    overall_mean = np.mean(K)
    K_centered = K - row_mean - col_mean + overall_mean
    return K_centered

def build_kernel(u, v, opt, gamma=1e-8, center=True):
    if opt["kernel_type"] == 'RBF':
        sq_dist = np.sum(u**2, axis=1)[:, None] + np.sum(v**2, axis=1)[None, :] - 2 * (u @ v.T)
        RBF = np.exp(-gamma * sq_dist)
        if center:
            RBF = center_kernel(RBF)
        return RBF
    elif opt["kernel_type"] == 'Linear':
        linear = u @ v.T
        if center:
            linear = center_kernel(linear)
        return linear
    elif opt["kernel_type"] == 'LinearRBF':
        linear = u @ v.T
        sq_dist = np.sum(u**2, axis=1)[:, None] + np.sum(v**2, axis=1)[None, :] - 2 * (u @ v.T)
        RBF = np.exp(-gamma * sq_dist)
        linearRBF = linear + RBF
        if center:
            linearRBF = center_kernel(linearRBF)
        return linearRBF
```

Figure 3: Kernel

1.1.3 PCA and Kernel PCA

PCA is a method that transforms data into a new coordinate system, such that the first largest variation of projected data is on the first coordinate (the first principal component), the second largest variation is on the second coordinate (the second principal component), and so on. In other words, we want to find

an orthogonal projection $W \in \mathbb{R}^{\times \urcorner}$, where the data $X \in \mathbb{R}^{\kappa \times}$ after projection $Z = XW$ will have maximum variance.

$$
\begin{aligned}
var(Z) =& \frac{1}{n} \sum (z_i - mean(Z))^T (z_i - mean(Z)) && (5)\\
=& \frac{1}{n} \sum ((x_i - mean(X)w_i)^T ((x_i - mean(X)w_i) && (6)\\
=& W^T cov(X) W := W^T S W && (7)
\end{aligned}
$$

By Rayleigh theorem, the optimal solution to $\arg max\ W^T S W\ s.t.\ W^T W = 1$ is given by the first k large eigenvectors of S.
Consider a function ϕ that maps data from data space to the feature space. Then S in Eq.7 becomes $\frac{1}{n}(\phi(X) - mean\phi(X))^T(\phi(X) - mean\phi(X)) = \frac{1}{n}\Phi(X)^T\Phi(X)$ and the objective function becomes

$$
\begin{aligned}
& \frac{1}{n}\Phi(X)^T\Phi(X)w = \lambda w && (8)\\
\Rightarrow & \frac{1}{n}\Phi(X)^T\Phi(X)\Phi(X)^T\alpha = \lambda\Phi(X)^T\alpha && (9)\\
\Rightarrow & \frac{1}{n}\Phi(X)\Phi(X)^T\Phi(X)\Phi(X)^T\alpha = \lambda\Phi(X)\Phi(X)^T\alpha && (10)\\
\Rightarrow & K^c K^c \alpha = \lambda n K^c \alpha,\ K^c = \Phi^T\Phi && (11)\\
\Rightarrow & K^c \alpha = \lambda n \alpha && (12)
\end{aligned}
$$

The optimal solution to α is given by the first k large eigenvectors of K^c.
To project new data X_{new}

$$\Phi(X_{new})W = \sum \alpha_i \Phi(x_i)\Phi(x_{new}) = \sum \alpha_i K(x_i, x_{new}) \quad (13)$$

```python
def pca(train, opt, kernel=False):
    if kernel:
        matrix = train
    else:
        # Compute covariance matrix
        matrix = np.cov(train.T)

    # Compute eigenvalues and eigenvectors
    eigenvalues, eigenvectors = np.linalg.eigh(matrix)

    # Find k eigenvectors corresponding to k largest eigenvalues
    sorted_indices = np.argsort(eigenvalues)[::-1]
    top_k_indices = sorted_indices[:opt["num_feature"]]
    W = eigenvectors[:, top_k_indices]

    # Normalized
    W = W / np.linalg.norm(W, axis=0, keepdims=True)
    return W
```

1.1.4 LDA nd Kernel LDA

In LDA we want to find a projection matrix W such that the between-class scatter after projection is maximized and the within-class scatter is minimized. The definition of between-class scatter and within-class scatter is as follow

- **within-class scatter**
  $S_w = \sum_{k=1}^{c} \sum_{x_i \in c_k} (x_i - \mu_k)(x_i - \mu_K)^T$
  , where μ_k is the mean of k-th class.

- **between-class scatter**
  $S_b = \sum_{k=1}^{c} N_k(\mu_k - \mu)(\mu_k - \mu)^T$
  , where N_k is the sample size of k-th class, μ is the total data mean.

Then the objective function is $\arg max\ J(W) = \frac{|W^T S_b W|}{|W^T S_w W|}$ and the optimal solution can be obtained by solving generalized eigenvalue problem $S_w^{-1} S_b W = \lambda W$.

```python
def lda(train, label, opt):
    data_size = train.shape[1]

    between_class = np.zeros((data_size, data_size), dtype=np.float64)
    within_class = np.zeros((data_size, data_size), dtype=np.float64)
    mean = np.mean(train, axis = 0) # Mean of all data

    for c in np.unique(label):
        X_c = train[label == c] # Data belonging to class c
        mean_c = np.mean(X_c, axis=0) # Class mean

        # Sum{(X_c - m_c)^t*(X_c - m_c)}
        within_class += (X_c - mean_c).T @ (X_c - mean_c)
        # Sum{n_c*(m_c - m)*(m_c - m)^t}
        diff = (mean_c - mean).reshape(-1, 1)
        between_class += len(X_c) * (diff @ diff.T)

    # Get first K largest eigenvectors of S_w^-1 * S_B
    S = np.linalg.pinv(within_class) @ between_class

    eigenvalues, eigenvectors = np.linalg.eigh(S)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    top_k_indices = sorted_indices[:opt["num_feature"]]
    W = eigenvectors[:, top_k_indices]

    # Normalized
    W = W / np.linalg.norm(W, axis=0, keepdims=True)
    return W
```

Figure 5: LDA

Consider a function ϕ that maps data from data space to the feature space. Then between-class scatter and within-class scatter become

- **within-class scatter in kernel space**
  $S_w^\phi = \sum_{k=1}^{c} \sum_{\phi(x_i) \in c_k} (\phi(x_i) - \mu_k^\phi)(\phi(x_i) - \mu_K^\phi)^T$

- **between-class scatter in kernel space**
  $S_b^\phi = \sum_{k=1}^{c} N_k (\mu_k^\phi - \mu^\phi)(\mu_k^\phi - \mu^\phi)^T$

and the objective function $J^\phi(W) = \frac{|{W^\phi}^T S_b^\phi W^\phi|}{|{W^\phi}^T S_w^\phi W^\phi|}$ and the optimal solution can be obtained by solving generalized eigenvalue problem
$S_w^{\phi^{-1}} S_b^\phi W^\phi = \lambda W^\phi$.

1.1.5 Reconstruction Eigenface

To reconstruction eigenface we compute $x' = xWW^T$ and reshape x' back to the image size (w', h')

```python
def reconstruction(W, X, width, height, opt, cols=5):
    data_size = len(X)
    num_reconstruction = opt["num_reconstruction"]

    # Randomly pick some images to show their reconstruction
    idx = np.random.choice(data_size, num_reconstruction, replace=False)

    fig = plt.figure(figsize=(10, 3))
    for i in range(len(idx)):
        data = X[idx[i]]
        data_reshaped = data.reshape(height, width)

        # Reconstruction x * W * W^T
        reconstructed_img = (data @ W @ W.T).reshape(height, width)
        reconstructed_img = normalize_img(reconstructed_img)
        img = np.hstack([data_reshaped, reconstructed_img])

        ax = fig.add_subplot(int(np.ceil(len(idx) / cols)), cols, i+1)
        ax.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.imshow(img, cmap='gray')

    plt.tight_layout()
    fig.savefig(os.path.join(opt["save_root"], f'{opt["task"]}_reconstructed.jpg'))
    plt.show()
```

Figure 6: Reconstruction Eigenface

1.1.6 Recognition

To perform face recognition, we project the training X and testing Y data using the projection matrix W obtained from PCA or LDA.
$X_{proj} = XW,\ Y_{proj} = YW$.
By projecting training and testing data through W, the dimensionality of the data is reduced, making subsequent classification tasks more efficient in the reduced space. For classification, we apply the K-Nearest Neighbor algorithm. For each test sample $y_{proj} \in Y_{proj}$, we compute the squared Euclidean distance between y_{proj} and X_{proj}, then select k training samples that closest to y_{proj} and the predicted class label for y_{proj} is determined by a majority vote among the class labels of the k nearest neighbors.

```python
def recognition(train, labelTr, test, labelTs, W, opt):
    X_W = train @ W
    Y_W = test @ W

    # K-NN
    error = 0
    K = opt["K-NN_k"]
    for i in range(len(test)):
        dist = np.zeros(len(train))
        for j in range(len(train)):
            # Distance between training data and testing data
            dist[j] = np.sum((X_W[j] - Y_W[i]) ** 2)

        nearest_neighbor = np.argsort(dist)[:K] # Find the K nearest neighbor
        k_nearest_labels = [labelTr[n] for n in nearest_neighbor]
        label_count = {} # Assign label
        for label in k_nearest_labels:
            if label in label_count:
                label_count[label] += 1
            else:
                label_count[label] = 1
        predict = max(label_count, key=label_count.get)

        if predict != labelTs[i]:
            error += 1
    accuracy_rate = 1 - (error / len(test))
    print(f'Task: {opt["task"]}, ACC: {accuracy_rate:.4f}, Error: {error}')
```

Figure 7: Recognition

1.2 Symmetric SNE and t-SNE

Symmetric SNE is a dimensionality reduction method designed to embed high-dimensional data into a lower-dimensional space. It preserves the neighborhood relationships between data points during the dimensionality reduction process. In the high-dimensional space, conditional probabilities p_{ij} are calculated for each data point using a Gaussian distribution

$$p_{ij} = \frac{exp(-\|x_i - x_j\|^2/(2\sigma^2))}{\sum_{l \neq k} exp(-\|x_l - x_k\|^2/(2\sigma^2))} \tag{14}$$

In the low-dimensional space, the goal is to make these similarity probabilities match the ones in the high-dimensional space as closely as possible

$$q_{ij} = \frac{exp(-\|y_i - y_j\|^2)}{\sum_{l \neq k} exp(-\|y_l - y_k\|^2)} \tag{15}$$

This is achieved by minimizing the KL divergence C between the two distributions P and Q and the final gradient formula is

$$\frac{\partial C}{\partial y_i} = 2 \sum_j (p_{ij} - q_{ij})(y_i - y_j) \tag{16}$$

t-SNE is an improved version of Symmetric SNE that uses a t-distribution to compute similarities in the low-dimensional space

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{l \neq k} (1 + \|y_l - y_k\|^2)^{-1}} \tag{17}$$

and the gradient become

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1} \tag{18}$$

This approach effectively addresses the "crowding problem," where points that are moderately distant in the high-dimensional space might not be properly represented in the low-dimensional space.

```
""" Modification 1 """
if method == 't-SNE':
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
elif method == 'sym-SNE':
    num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))
```

Figure 8: Modified Similarity Code

In the main function(Fig. 10), we perform symmetric SNE and t-SNE with perplexity values $p \in [10, 20, 30, 40, 50]$. We make the clustering process into a GIF through **save_procedure** and **make_video** function(Fig. 11) and plot the distribution of pairwise similarities in both high-dimensional space and low-dimensional space through **plot_pairwise_similarities** function(Fig 12).

```
""" Modification 2 """
if method == '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)
elif method == 'sym-SNE':
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Figure 9: Modified Gradient Code

```
if __name__ == "__main__":
    method = 't-SNE' # t-SNE, sym-SNE
    save_root = os.path.join('results_SNE', method)

    perplexity = [10, 20, 30, 40, 50]

    X = np.loadtxt("mnist2500_X.txt")
    labels = np.loadtxt("mnist2500_labels.txt")

    for p in perplexity:
        print(f"Perplexity: {p}")

        save_dir = os.path.join(save_root, f"p_{p:03d}")
        os.makedirs(save_dir, exist_ok=True)
        Y, P, Q = sne(X, labels, 2, 50, p, method, save_dir)

        # Plot pairwise similarities
        plot_pairwise_similarities(P, Q, labels, p, method, save_dir)

        # Mack video
        make_video(save_dir, method, p, remove=True)
```

Figure 10: Main Function of tSNE

```
def save_procedure(Y, labels, iter, save_dir):
    plt.figure()
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.axis('off')
    plt.savefig(os.path.join(save_dir, f"{(iter):04d}_procedure.png"))
    plt.close()

def make_video(fig_path, method, perplexity, remove=False):
    fig_list = sorted([os.path.join(fig_path, f) for f in os.listdir(fig_path) if f.endswith(('procedure.png'))])

    imgs = [Image.open(i) for i in fig_list]
    imgs[0].save(os.path.join(fig_path, f"{method}_p_{perplexity:03d}.gif"),
                 save_all=True,
                 append_images=imgs[1:],
                 duration=250,
                 loop=0)
    if remove:
        for f in os.listdir(fig_path):
            if f.endswith(('procedure.png')):
                os.remove(os.path.join(fig_path, f))
```

Figure 11: Save Procedure

```python
def plot_pairwise_similarities(P, Q, labels, perplexity, method, save_dir):
    # Sort
    idx = labels.argsort()
    P = -np.log(P[:, idx][idx])
    Q = -np.log(Q[:, idx][idx])

    # High-D similarity
    plt.figure()
    sns.heatmap(P, vmin=np.min(P), vmax=np.max(P), cmap='BuGn', cbar=True)
    plt.title("High-Dimensional Similarity (P)")
    plt.savefig(os.path.join(save_dir, f"{method}_p_{perplexity:03d}_high-d_similarity.png"))
    plt.close()

    # Low-D similarity
    plt.figure()
    sns.heatmap(Q, vmin=np.min(Q), vmax=np.max(Q), cmap='BuGn', cbar=True)
    plt.title("Low-Dimensional Similarity (Q)")
    plt.savefig(os.path.join(save_dir, f"{method}_p_{perplexity:03d}_low-d_similarity.png"))
    plt.close()
```

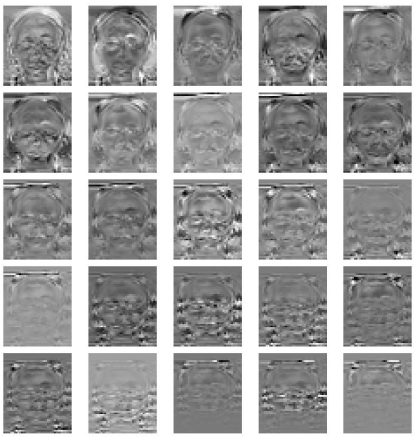
Figure 12: Plot Pairwise Similarities

2 Experiments and Discussion

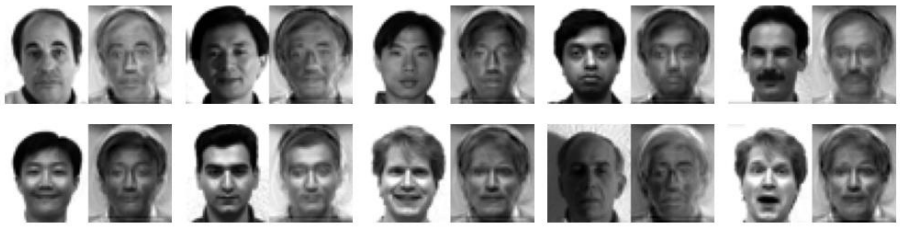
2.1 Kernel Eigenface

Fig.13a and Fig.13b show the first 25 eigenfaces and fisherfaces of PCA and LDA, and Fig.14a and Fig.14b show the 10 reconstructions of randomly selected images.

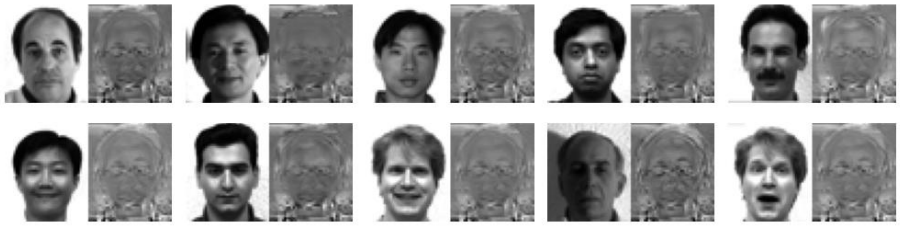
(a) PCA Eigenface



(b) LDA Fisherfaces



(a) PCA Reconstruction



(b) LDA Reconstruction

Table 1 shows the performance of PCA and LDA with different kernels. We found that RBF performed best when using the kernel method, but it did not significantly improve the accuracy compared to not using the kernel method. Compared with PCA, LDA does not have a higher accuracy rate regardless of whether kernel methods are used or not. It may be because the label class of training data is 15 but we embedded it into the 25-D space.

PCA	None	Linear	RBF	LinearRBF
Acc. rate	0.8667	0.7667	0.8333	0.7667
LDA	None	Linear	RBF	LinearRBF
Acc. rate	0.8333	0.7667	0.8000	0.7667

Table 1: Performance of PCA and LDA

2.2 t-SNE

Fig. 15 shows the embedding results of symmetric SNE and t-SNE. We observed that, due to the asymmetry of the KL divergence, SSNE results in an overcrowded point distribution, leading to the "crowding problem." This issue is alleviated in t-SNE by using a heavy-tailed Student-t distribution for the low-dimensional space, which allows points to spread out more effectively. As the perplexity increases, t-SNE performs clearer cluster separation and better preservation of local structures, while S-SNE continues to exhibit dense point distributions with less distinct boundaries between clusters.
Fig. 16 shows the pairwise similarity of both methods, we clearly found that the color of similarity of t-SNE is lighter than that of S-SNE, indicating a more balanced distribution of pairwise similarities in the low-dimensional embedding.
The GIF images of the output result will be named in the following way and stored in the **Results_GIF** folder.
[METHOD]_p_[PERPLEXITY].gif
For example, sym-SNE_p_030.gif means the result of symmetric SNE using perplexity $p = 30$.

3 Observations

- **The meaning of eigenfaces**
  Instead of analyzing the raw pixel data (high-dimensional space), eigenfaces reduce the data to a low-dimensional subspace, focusing on the most important components. Any face image can be approximated by a weighted sum of eigenfaces, where the weights representing the coordinates in the reduced-dimensional space. This approach significantly reduces the number of features, making computations more efficient. However, eigenfaces are highly affected by variations in lighting and face orientation, and some fine details may be lost during dimensionality reduction.

- **The relationship between perplexity and σ in S-SNE and t-SNE**
  In S-SNE and t-SNE, σ is a parameter that controls the width of the Gaussian distribution for each data point x_i. We want to adjust σ so that the entropy H of the probability distribution matches the target perplexity. In other words, σ determines the shape of the similarity distribution, and it satisfy $2^{H(P_i)} = Perplexity$. Start with an initial σ value, then compute the corresponding entropy and compare with the target perplexity, If $2^{H(P_i)}$ is too large, decrease σ; otherwise increase σ. Repeat the adjustment until $2^{H(P_i)} \approx Perplexity$. A smaller perplexity value means the neighborhood is smaller, and t-SNE focuses more on the local structure of the data, resulting in less emphasis on global structures such as cluster boundaries. While a larger perplexity making t-SNE consider more neighbors and thus emphasizing the global structure, but blur some of the finer local details.

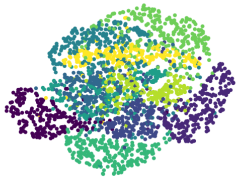
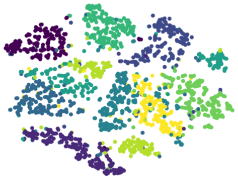
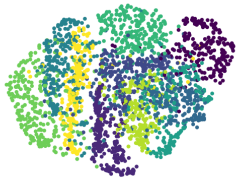
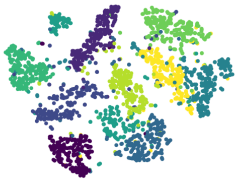
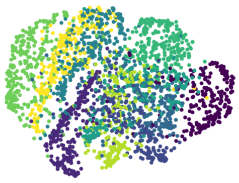
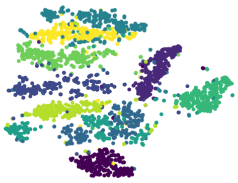
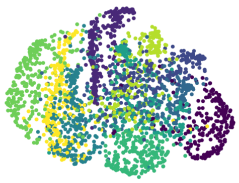
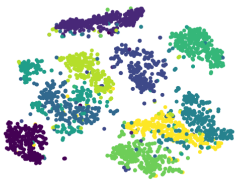
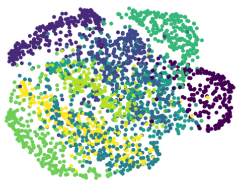
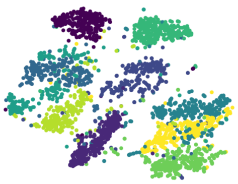
Perplexity	SSNE	t-SNE
$p = 10$		
$p = 20$		
$p = 30$		
$p = 40$		
$p = 50$		

Figure 15: Results of t-SNE and symmetric SNE

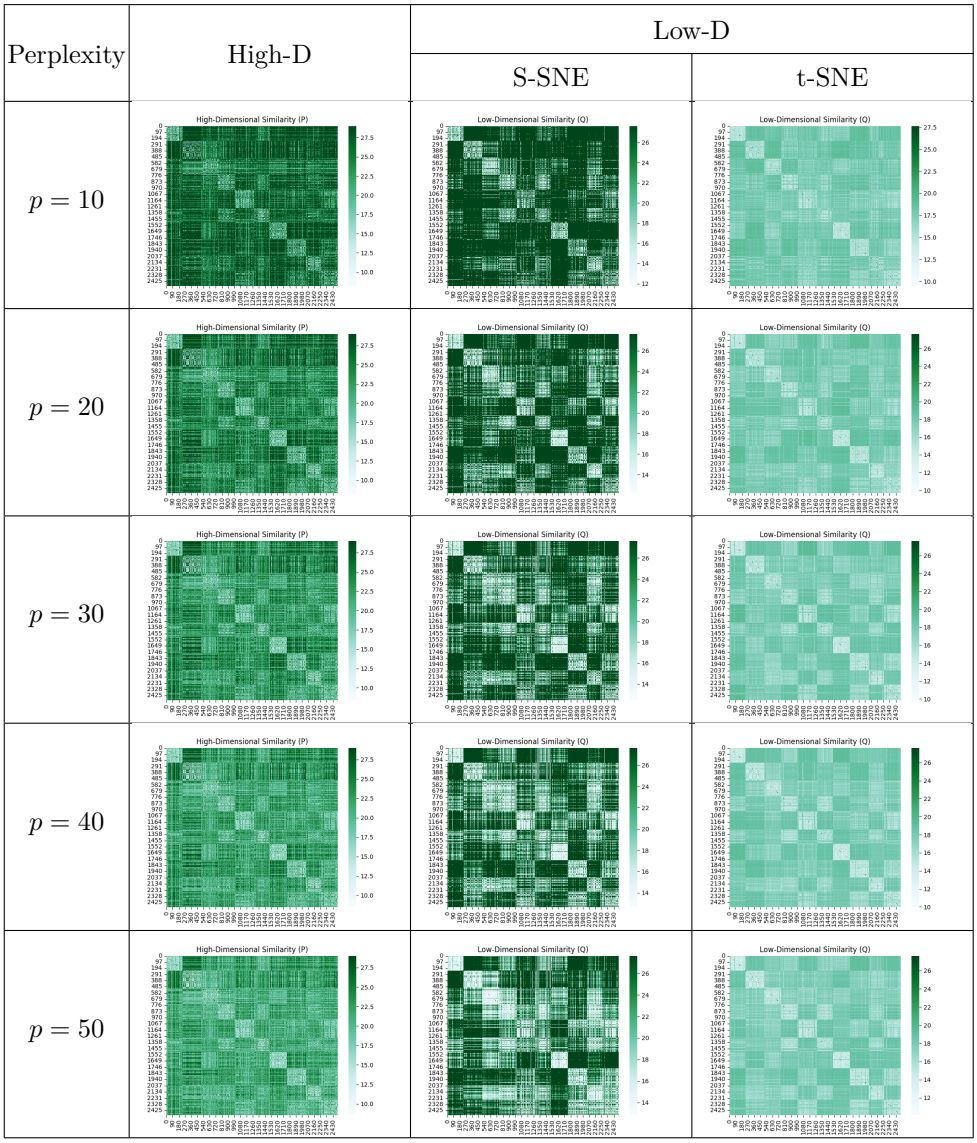


Figure 16: Pairwise Similarities of t-SNE and symmetric SNE