
Machine Learning Homework 7 Report

陳以瑄 (ID: 313551047)

1 Code & Explanation

1.1 Kernel Eigenfaces

1.1.1 Part 1: PCA & LDA

1) Prepare data

Load 135 training images and 30 testing images, and extract the labels (subject IDs) from the filenames. The images are resized from 231×195 to one-third of their original dimensions to simplify computations.

```
1 scalar = 3
2 img_h = 231//scalar
3 img_w = 195//scalar
4 train_num = 135
5 test_num = 30
```

```
1 train_imgs, train_labels = LoadData('./Yale_Face_Database/Training/')
2 test_imgs, test_labels = LoadData('./Yale_Face_Database/Testing/')
```

```
1 def LoadData(path):
2     imgs = []
3     labels = []
4     for file in os.listdir(path):
5         labels.append(int(file[7:9]))
6         img = LoadPmg(os.path.join(path, file))
7         img = Resize(img)
8         imgs.append(img)
9     imgs = np.array(imgs)
10    labels = np.array(labels)
11    return imgs, labels
```

```
1 def LoadPmg(path):
2     with open(path, 'rb') as file:
3         # the first 4 lines are
4         # 1. magic number P5 = binary file
5         # 2. comment line
6         # 3. width & height = 195* 231
7         # 4. maximum value
8         file.readline()
9         file.readline()
10
11        line = file.readline().split()
12        w = int(line[0])
13        h = int(line[1])
14
15        file.readline()
16
17        img = np.zeros((h, w))
18        for r in range(h):
19            for c in range(w):
20                img[r][c] = ord(file.read(1))
21
22        return img
```

```

1 def Resize(img):
2     new_img = np.zeros((img_h, img_w))
3     for r in range(img_h):
4         for c in range(img_w):
5             val = 0
6             for i in range(scalar):
7                 for j in range(scalar):
8                     val+=img[r*scalar+i][c*scalar+j]
9             new_img[r][c] = val//(scalar*scalar)
10    new_img = new_img.reshape(-1)
11    return new_img

```

2) Principal Component Analysis (PCA)

The goal of PCA is to find an orthogonal projection matrix W such that the projected data $z = Wx$ has the maximum variance. This is achieved by maximizing the $Cov(z) = WSW^T$, where $S = Cov(x)$. The solution for W is constructed from the k largest eigenvectors of S .

To achieve this, I first calculate the covariance matrix S of the input data X . Next, I solve the eigenvalue problem, sort the eigenvectors in descending order of their corresponding eigenvalues, and select the top 25 eigenvectors to form W .

```

1 def PCA(X):
2     cov = np.cov(X.T)
3     eigenvalue, eigenvector = np.linalg.eigh(cov)
4     sorted_idx = np.argsort(eigenvalue)[::-1]
5     eigenvector = eigenvector[:,sorted_idx]
6     W = eigenvector[:, :k_eigen]
7     # whitening
8     for i in range(k_eigen):
9         W[:, i] /= np.linalg.norm(W[:, i])
10    return W

```

3) Linear Discriminant Analysis (LDA)

The goal of LDA is to find a projection matrix W such that the projected data $z = Wx$ maximally separates the classes. This is achieved by maximizing the between-class scatter while minimizing the within-class scatter. The mathematical definitions of the scatter matrices are as follows:

within – class scatter : $S_W = \sum_{j=1}^k S_j$, $S_j = \sum_{i \in C_j} (x_i - m_j)(x_i - m_j)^T$, $m_j = \frac{1}{n_j} \sum_{i \in C_j} x_i$

between – class scatter : $S_B = \sum_{j=1}^k S_{B_j} = \sum_{j=1}^k n_j(m_j - m)(m_j - m)^T$, $m = \frac{1}{n} \sum x$

The solution for W is constructed by solving the generalized eigenvalue problem $S = S_W^{-1}S_B$, where the eigenvectors corresponding to the largest eigenvalues are selected.

To achieve this, I first compute the within-class scatter matrix S_W and the between-class scatter matrix S_B using the input data X and their class labels. Next, I solve the eigenvalue problem, sort the eigenvectors in descending order of their corresponding eigenvalues, and select the top 25 eigenvectors to form W . Finally, I perform whitening on W to normalize the scaling differences across its components.

```

1 def LDA(X,y):
2     dim = img_h* img_w
3     m = np.mean(X,axis=0)
4     S_B = np.zeros((dim,dim))
5     S_W = np.zeros((dim,dim))
6
7     for j in tqdm(range(1,16), desc="Class:"):
8         Xj = X[y==j]
9         mj = np.mean(Xj, axis=0)
10
11         Sj = (Xj - mj).T @ (Xj - mj)
12         S_W+=Sj
13
14         diff = (mj - m).reshape(-1,1)
15         SBj = diff @ diff.T
16         S_B += len(Xj)*SBj
17
18     S = np.linalg.pinv(S_W) @ S_B
19     eigenvalue, eigenvector = np.linalg.eigh(S)
20     sorted_idx = np.argsort(eigenvalue)[::-1]
21     eigenvector = eigenvector[:,sorted_idx]
22     W = eigenvector[:, :k_eigen]
23     return W

```

4) Visualize the eigenfaces/ fisherfaces

In PCA, each column of the solution matrix W corresponds to an eigenface. Similarly, in LDA, each column of the solution matrix W corresponds to a fisherface.

```

1 def PrintFace(W, title):
2     fig, axes = plt.subplots(5, 5, figsize=(6,6))
3     axes = axes.flatten()
4     for i in range(k_eigen):
5         img = W[:,i].reshape(img_h,img_w)
6         axes[i].imshow(img, cmap='gray')
7         axes[i].axis('off')
8     plt.savefig(f"./output/{title}.png")
9     plt.tight_layout()
10    plt.show()

```

5) Reconstruction

Given a projection matrix W , the reconstruction formula is: $x_{rec} = zW^T = xWW^T$.

The process is as follows: it first projects the original image x onto the lower-dimensional space using W to obtain z , and then maps it back to the original space via W^T .

```

1 def Reconstruct(W,train,title):
2     idx = np.random.choice(train_num, 10, replace = False)
3     fig, axes = plt.subplots(2, 10, figsize=(12,3))
4     for i in range(10):
5         img = train[idx[i]].reshape(img_h,img_w)
6         axes[0][i].imshow(img, cmap='gray')
7         axes[0][i].axis('off')
8
9         img = train[idx[i]]
10        rec_img = img @ W @ W.T
11        rec_img = rec_img.reshape(img_h,img_w)
12        axes[1][i].imshow(rec_img, cmap='gray')
13        axes[1][i].axis('off')
14    plt.savefig(f"./output/re_{title}.png")
15    plt.tight_layout()
16    plt.show()

```

1.1.2 Part 2: KNN face recognition

To perform KNN-based face recognition, the process is as follows: first, project both the training and test images using W . Then, use KNN to classify the test images, where the number of neighbors $k = 5$. Finally, output the classification accuracy.

```
1 def Prediction(W,train_imgs, train_labels, test_imgs, test_labels):
2     train_Z = train_imgs @ W
3     test_Z = test_imgs @ W
4     acc = test_num
5     for i in range(test_num):
6         dist = np.zeros(train_num)
7         for j in range(train_num):
8             dist[j] = np.sum((test_Z[i]-train_Z[j])**2)
9         nei_idx = np.argsort(dist)[:k_neighbor]
10
11         class_counts = {}
12         for label in train_labels[nei_idx]:
13             if label in class_counts:
14                 class_counts[label] += 1
15             else:
16                 class_counts[label] = 1
17
18         pred = max(class_counts, key=class_counts.get)
19         if(test_labels[i]!=pred):
20             acc-=1
21     print(f"accuracy: {(acc/test_num)*100:.2f}% ({acc}/{test_num})")
```

1.1.3 Part 3: Kernel PCA & LDA

To perform kernel PCA and LDA, I modified the following parts:

1) Centered the data

To simplify the computation of the eigenvalue problem of the covariance matrix in the feature space, the data first needs to be centered.

```
1 mean = np.mean(train_imgs, axis = 0)
2 C_train = train_imgs - mean
3 C_test = test_imgs - mean
```

2) Compute kernel

I used both the linear kernel and the RBF kernel. The formulas are:

Linear : $K = XX'^T$

RBF : $K = e^{-\gamma D}$, where $D = \|x_i\|^2 + \|x_j\|^2 - 2(x_i \cdot x_j)$

```
1 def Linear_K(X1,X2):
2     return X1 @ X2.T
```

```
1 def RBF_K(X1,X2):
2     square_X1 = np.sum(X1 ** 2, axis=1).reshape(-1, 1)
3     square_X2 = np.sum(X2 ** 2, axis=1)
4     d = square_X1 - 2 * X1 @ X2.T + square_X2
5     k = np.exp(-gamma * d)
6     return k
```

3) Kernel PCA

To perform kernel PCA, first project the data into the feature space using the specified kernel (linear or RBF). Then, compute the covariance matrix on the kernel matrix rather than on the original data. The following steps are the same as in the original PCA.

```

1 def PCA(X, kernel = None):
2     if kernel == 'linear':
3         X = Linear_K(X,X)
4     elif kernel == 'rbf':
5         X = RBF_K(X,X)
6
7     cov = np.cov(X.T)
8     eigenvalue, eigenvector = np.linalg.eigh(cov)
9
10    sorted_idx = np.argsort(eigenvalue)[::-1]
11    eigenvector = eigenvector[:,sorted_idx]
12    W = eigenvector[:, :k_eigen]
13    # whitening
14    for i in range(k_eigen):
15        W[:, i] /= np.linalg.norm(W[:, i])
16
17    return W, X

```

4) Kernel LDA

Similar to kernel PCA, to perform kernel LDA, we first project the data into the feature space by computing the kernel. In kernel LDA, the eigenproblem is:

$$K = [\phi(x_1), \phi(x_2), \dots, \phi(x_n)]$$

$$m^\phi = \frac{1}{n} \sum \phi(x) = 0 \text{ (centered)}$$

$$m_j^\phi = \frac{1}{n_j} \sum_{i \in C_j} \phi(x_i)$$

$$\text{within - class scatter : } S_W^\phi = \sum_{j=1}^k \frac{1}{n} S_j, \text{ where } S_j = \sum_{i \in C_j} (\phi(x_i) - m_j^\phi)(\phi(x_i) - m_j^\phi)^T$$

$$\text{between - class scatter : } S_B^\phi = \sum_{j=1}^k \frac{n_j}{n} (m_j^\phi - m^\phi)(m_j^\phi - m^\phi)^T = \sum_{j=1}^k \frac{n_j}{n} m_j^\phi (m_j^\phi)^T$$

$$m_j^\phi = \frac{1}{n_j} \sum_{i \in C_j} \phi(x_i) = \frac{1}{n_j} [\phi(x_{j,1}), \phi(x_{j,2})] \mathbf{1}_{n_j} = \frac{1}{n_j} K_j^T \mathbf{1}_{n_j}$$

$$m_j^\phi (m_j^\phi)^T = \frac{1}{n_j^2} K_j^T \mathbf{1}_{n_j} \mathbf{1}_{n_j}^T K_j = \frac{1}{n_j} K_j^T Z K_j$$

$$Z = \frac{1}{n_j} \mathbf{1}_{n_j} \mathbf{1}_{n_j}^T = \frac{1}{n_j} \mathbf{1}_{n_j \times n_j}$$

$$S_B^\phi = \sum_{j=1}^k \frac{n_j}{n} m_j^\phi (m_j^\phi)^T = \sum_{j=1}^k \frac{n_j}{n} \frac{1}{n_j} K_j^T Z K_j = \frac{1}{n} \sum_{j=1}^k K_j^T Z K_j = \frac{1}{n} K^T Z K$$

$$\begin{aligned}
 S_W^\phi &= \frac{1}{n} \sum_{j=1}^k \sum_{i \in C_j} (\phi(x_i) - m_j^\phi)(\phi(x_i) - m_j^\phi)^T = \frac{1}{n} \sum_{j=1}^k [\phi(x_{j,1}), \dots, \phi(x_{j,n_j})][\phi(x_{j,1}), \dots, \phi(x_{j,n_j})]^T \\
 &= \frac{1}{n} \sum_{j=1}^k K_j^T K_j = \frac{1}{n} K^T K
 \end{aligned}$$

Therefore, the eigenproblem become $\lambda K K v = K Z K v$.

```

1 def LDA(X,y, kernel = None):
2     if kernel == 'linear':
3         K = Linear_K(X,X)
4         Z = np.ones((train_num, train_num)) / 9
5         S_B = K @ Z @ K
6         S_W = K @ K
7     elif kernel == 'rbf':
8         K = RBF_K(X,X)
9         Z = np.ones((train_num, train_num)) / 9
10        S_B = K @ Z @ K
11        S_W = K @ K
12
13    S = np.linalg.pinv(S_W) @ S_B
14    eigenvalue, eigenvector = np.linalg.eigh(S)
15
16    sorted_idx = np.argsort(eigenvalue)[::-1]
17    eigenvector = eigenvector[:,sorted_idx]
18    W = eigenvector[:, :k_eigen]
19    # whitening
20    for i in range(k_eigen):
21        W[:, i] /= np.linalg.norm(W[:, i])
22
23    return W, K

```

5) Kernel prediction

To make predictions, we first compute the kernel values between the test data and the training data. After that, the prediction process follows the same steps as in the original KNN method.

```

1 def Prediction(W,train_imgs, train_labels, test_imgs, test_labels, train_K = None, kernel = None):
2     if kernel == 'linear':
3         test_imgs = Linear_K(test_imgs, train_imgs)
4         train_imgs = train_K
5     elif kernel == 'rbf':
6         test_imgs = RBF_K(test_imgs, train_imgs)
7         train_imgs = train_K
8     train_Z = train_imgs @ W
9     test_Z = test_imgs @ W
10
11    acc = test_num
12    for i in range(test_num):
13        dist = np.zeros(train_num)
14        for j in range(train_num):
15            dist[j] = np.sum((test_Z[i]-train_Z[j])**2)
16        nei_idx = np.argsort(dist)[:k_neighbor]
17
18        class_counts = {}
19        for label in train_labels[nei_idx]:
20            if label in class_counts:
21                class_counts[label] += 1
22            else:
23                class_counts[label] = 1
24        pred = max(class_counts, key=class_counts.get)
25        if(test_labels[i]!=pred):
26            acc-=1
27    print(f"accuracy: {(acc/test_num)*100:.2f}% ({acc}/{test_num})")

```

1.2 t-SNE

1.2.1 Part 1: Symmetric SNE

Difference 1: Low-dimensional distribution

	Distribution	Formula
tSNE	student-t	$q_{ij} = \frac{(1 + \ y_i - y_j\ ^2)^{-1}}{\sum_{k \neq l} (1 + \ y_k - y_l\ ^2)^{-1}}$
SNE	gaussian	$q_{ij} = \frac{e^{-\ y_i - y_j\ ^2}}{\sum_{k \neq l} e^{-\ y_k - y_l\ ^2}}$

```
41 # Compute pairwise affinities
42 sum_Y = np.sum(np.square(Y), 1)
43 num = -2. * np.dot(Y, Y.T)
44
45 # tSNE: student t distribution
46 # SNE: normal distribution
47 if mode == 'tsne':
48     num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
49 else:
50     num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
51
52 num[range(n), range(n)] = 0.
53 Q = num / np.sum(num)
54 Q = np.maximum(Q, 1e-12)
```

Difference 2: Gradient

$$\begin{aligned} \text{tSNE} \quad & \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1} \\ \text{SNE} \quad & \sum_j (p_{ij} - q_{ij})(y_i - y_j) \end{aligned}$$

```
56 # Compute gradient
57 PQ = P - Q
58 for i in range(n):
59     if mode == 'tsne':
60         dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
61     else:
62         dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

1.2.2 Part 2: Visualize the embedding

For 50 iterations, plot the embedding and save the results. After finishing the whole process, compile these plots into a GIF.

```
1 def Plot(Y, perplexity, labels, iter_, mode):
2     pylab.figure()
3     pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
4     if mode == 'tsne':
5         pylab.title(f't-SNE, perplexity: {perplexity}, iteration: {iter_}')
6     else:
7         pylab.title(f'SNE, perplexity: {perplexity}, iteration: {iter_}')
8
9     output_path = f'./result/{mode}/perplexity{perplexity}/{iter_}.png'
10    pylab.savefig(output_path)
```

```

1 def PNGtoGIF(perplexity, mode):
2     img_path = f'./result/{mode}/perplexity{perplexity}/'
3     output_path = './result/gif/'
4
5     image_files = [f for f in os.listdir(img_path) if f.endswith('.png')]
6     image_files.sort(key=lambda x: int(os.path.splitext(x)[0]))
7     images = []
8
9     for image_file in image_files:
10        image_path = os.path.join(img_path, image_file)
11        img = Image.open(image_path)
12
13        img = img.convert("RGBA")
14        background = Image.new("RGBA", img.size, (255, 255, 255, 255))
15        background.paste(img, (0, 0), img)
16        images.append(background.convert("RGB"))
17
18    if images:
19        gif_path = os.path.join(output_path, f"{mode}_perplexity{perplexity}.gif")
20        images[0].save(gif_path, save_all=True, append_images=images[1:], duration=500, loop=0)
21        print(f"GIF created: {gif_path}")
22    else:
23        print(f"No .png images found in {c_folder_path}")

```

1.2.3 Part 3: Pairwise similarity

I used a heatmap to display the pairwise similarity. Since the values were very small and difficult to compare, I applied a log transformation to the probability values. I sorted the results by index to group points from the same class together. Then, I plotted the heatmaps for both the low-dimensional and high-dimensional data. To facilitate comparison, I ensured that the value range for both plots was the same.

```

1 def PlotSimilarity(P, Q, labels, perplexity, mode):
2     output_path = './result/sim/'
3     png_path = os.path.join(output_path, f"{mode}_perplexity{perplexity}.png")
4
5     fig, ax = plt.subplots(1, 2, figsize=(10, 4.5))
6
7     idx = np.argsort(labels)
8     sortP = P[:, idx][idx]
9     sortQ = Q[:, idx][idx]
10    v_min = min(np.min(np.log(sortP + 1e-12)), np.min(np.log(sortQ + 1e-12)))
11    v_max = max(np.max(np.log(sortP + 1e-12)), np.max(np.log(sortQ + 1e-12)))
12
13    ax[0].set_title(f'{mode}(p={perplexity}) Similarity in High-Dim')
14    cax1 = ax[0].imshow(np.log(sortP + 1e-12), cmap='hot', aspect='auto', vmin=v_min, vmax=v_max)
15    fig.colorbar(cax1, ax=ax[0], shrink=0.7)
16
17    ax[1].set_title(f'{mode}(p={perplexity}) Similarity in Low-Dim')
18    cax2 = ax[1].imshow(np.log(sortQ + 1e-12), cmap='hot', aspect='auto', vmin=v_min, vmax=v_max)
19    fig.colorbar(cax2, ax=ax[1], shrink=0.7)
20
21    fig.tight_layout()
22    plt.savefig(png_path, bbox_inches='tight')
23    plt.show()

```

1.2.4 Part 4: Perplexity

I tried five perplexity values: 5, 15, 30, 50, 100.

```



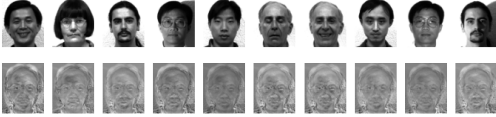
1 perplexities = [5,15,30,50,100]
2 for perplexity in perplexities:
3     SNE_TSNE(X, labels, 2, 50, perplexity, mode)

```


2 Experiments & Discussion

2.1 Kernel Eigenfaces

2.1.1 Part 1

	Eigenfaces/ Fisherfaces	Reconstruction faces
PCA		
LDA		

Discussion

Eigenface more effectively captures facial contours compared to Fisherface. Consequently, from a human visual perspective, PCA-reconstructed faces more closely resemble the original images.

2.1.2 Part 2

The number of neighbors $k=5$.

	Accuracy
PCA	90.00% (27/30)
LDA	93.33% (28/30)

2.1.3 Part 3

The number of neighbors $k=5$. The γ for the RBF kernel $= 1 \times 10^{-8}$.

	kernel	Accuracy
PCA	Linear	83.33% (25/30)
	RBF	76.67% (23/30)
LDA	Linear	80.00% (24/30)
	RBF	80.00% (24/30)

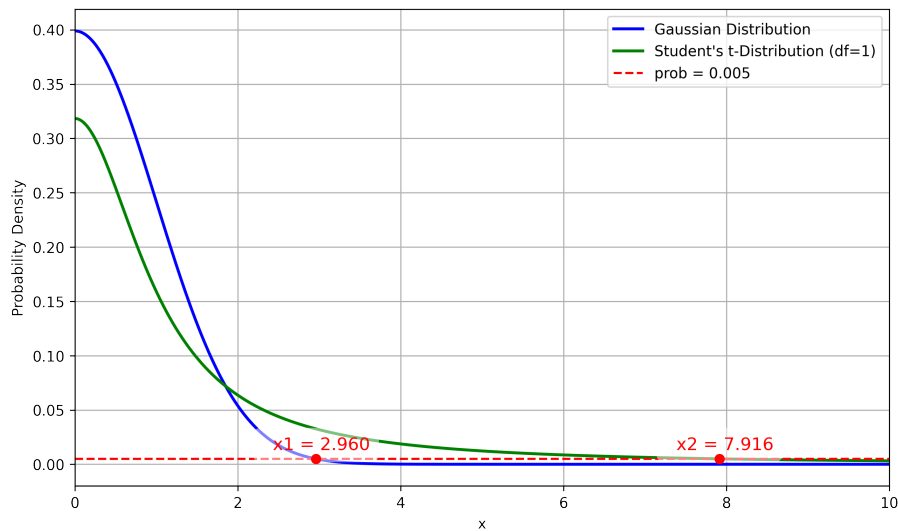
Discussion

Although Eigenface captures facial contours more clearly than Fisherface from a human visual perspective, the performance difference between the two in recognition tasks is not significant. From the results in Part 2, LDA slightly outperforms PCA (accuracy: 93.33% > 90.00%). In Part 3, we can observe that applying either a linear kernel or an RBF kernel does not seem to improve performance for this task. Both PCA and LDA kernel versions perform worse than their original versions, with PCA showing better results using the linear kernel compared to the RBF kernel. This outcome might be due to the original features already capturing the relevant information effectively, making the additional complexity introduced by the kernel unnecessary.

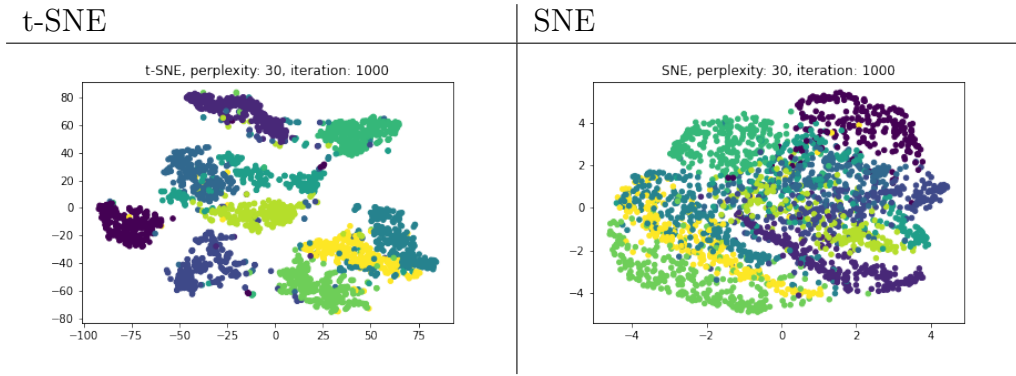
2.2 t-SNE

2.2.1 Part 1

It was mentioned in Section 1.2.1, that the primary distinction between symmetric SNE and t-SNE lies in how they model the probability distribution of similarities between two data points in the low-dimensional space. While SNE uses a Gaussian distribution, t-SNE employs a Student's t-distribution. This difference arises because t-SNE aims to address the "crowding problem" inherent in SNE, which is caused by the thin-tailed nature of the Gaussian distribution. As shown in the plot below, the Gaussian distribution can assign a small probability to pairs of points that are not very far apart. For instance, a probability of 0.005 can be achieved with a distance of 2.960, leading to a compact layout in the low-dimensional space. In contrast, t-SNE, with its heavier tails, uses the Student's t-distribution, which requires a larger distance—7.916 in this case—to achieve the same probability, thereby alleviating the crowding problem.



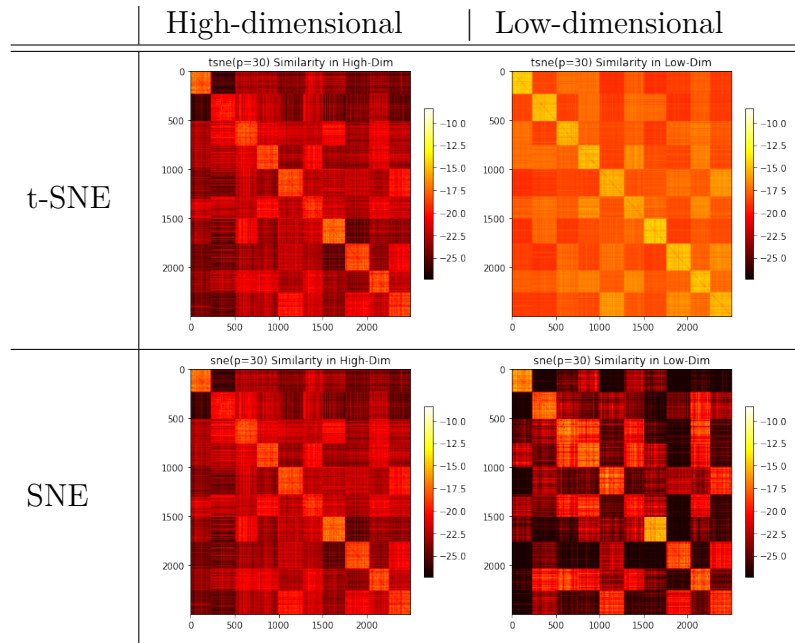
2.2.2 Part 2



Discussion

Comparing the embeddings of t-SNE and SNE with a perplexity of 30, it is evident that SNE suffers from the crowding problem. The range of each axis in SNE is limited to $[-5, 5]$, whereas t-SNE expands the range to $[-100, 100]$, resulting in better separation of data points from different classes.

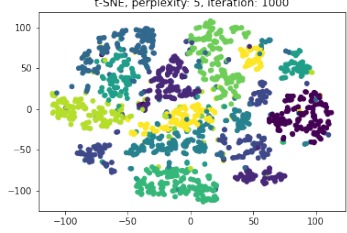
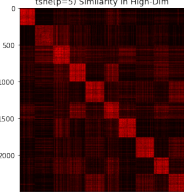
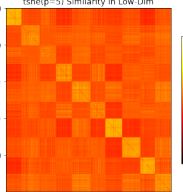
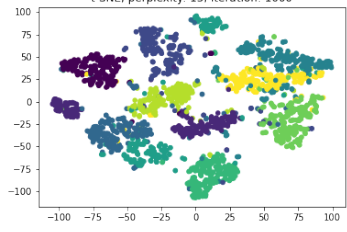
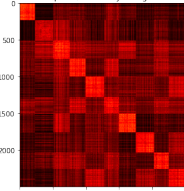
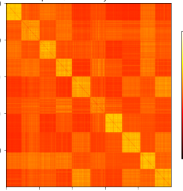
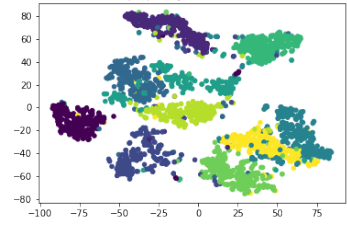
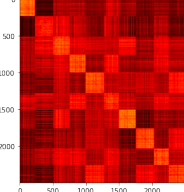
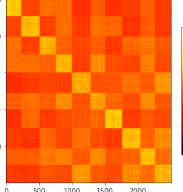
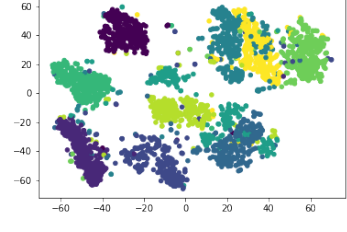
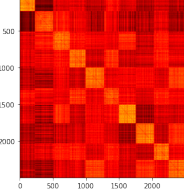
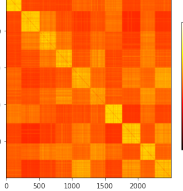
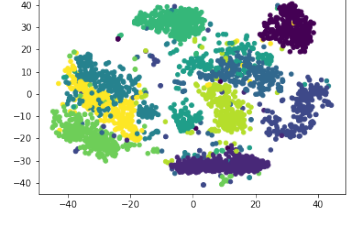
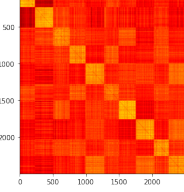
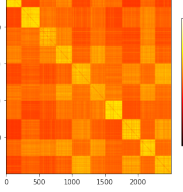
2.2.3 Part 3

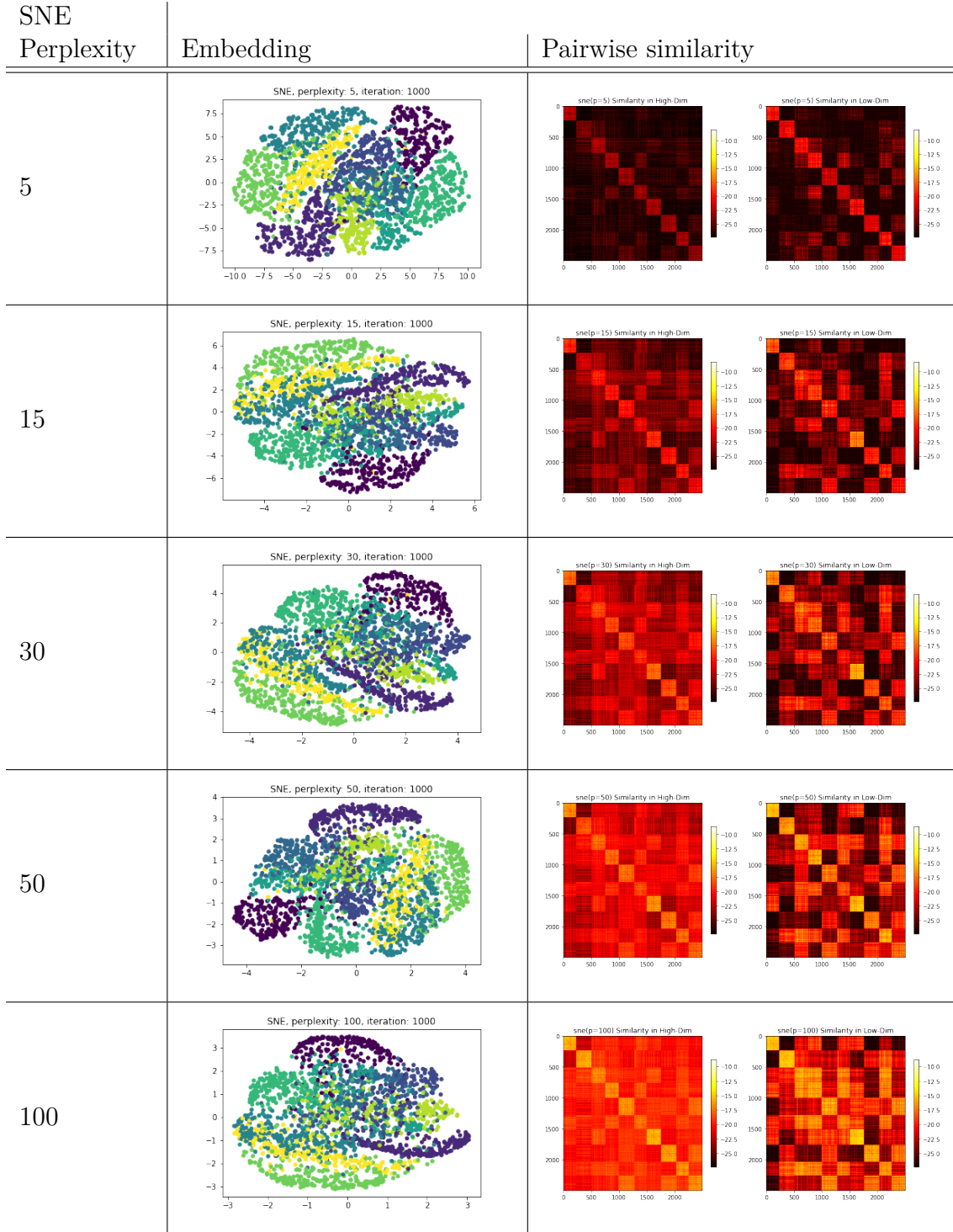


Discussion

In the low-dimensional space, while t-SNE generally produces higher probabilities (with colors closer to yellow), it displays significantly larger values for points within the same class, as reflected in the ten distinct yellow blocks along the diagonal. In contrast, SNE exhibits lower probabilities overall, especially between different classes, where many very low values are observed (with numerous dark blocks). However, SNE only forms nine clearly defined regions along the diagonal, indicating challenges in distinguishing classes 2 and 3. Additionally, within class 5, the probabilities are generally lower (with colors closer to orange-red). This contrast highlights t-SNE's stronger focus on preserving local similarity.

2.2.4 Part 4

t-SNE Perplexity	Embedding	Pairwise similarity	
5			
15			
30			
50			
100			



Discussion

Similar to part 2, when comparing the embeddings of t-SNE and SNE, we can see that t-SNE separates points from different classes more clearly. In contrast, SNE clusters points from the same class in nearby regions, but points from different classes are often very close or even overlapping.

When looking at different perplexity values, we notice that for both t-SNE and SNE, increasing the perplexity reduces the axis range. In t-SNE, smaller perplexity values cause greater distances between neighboring points, which leads to visible white gaps. Additionally, points from the same class may be scattered in small groups across different

areas. For example, with perplexity = 5, the yellow points form two small clusters. In contrast, with larger perplexity values, neighboring points tend to group more closely together.

When comparing the pairwise similarity heatmap in the high-dimensional space, we can see that only the similarity within the same class is relatively high, while the similarity between different classes is very low. For example, with perplexity = 5, the heatmap is mostly black, with only 10 red blocks along the diagonal. As the perplexity increases, the overall similarity values go up (with lighter colors), especially especially for the similarity across different classes.

Similar to the results in part 3, in the low-dimensional space, t-SNE generally produces higher overall values. The results also show that perplexity values of 15 and 30 perform better. When perplexity = 5, the similarity within the same class is relatively low, so the difference between similarity within classes and across classes is small, making it difficult to distinguish them. However, with perplexity = 15 and 30, the similarity within the same class is much higher than the similarity across classes, resulting in 10 distinct blocks that are clearly visible along the diagonal. When perplexity increases to 50 or 100, the same-class similarity remains higher along the diagonal, but the similarity between different classes also increases, making it harder to differentiate between them.

In SNE, the performance is more sensitive to the perplexity value. It only performs well at perplexity = 5, where, although the overall similarity values are low, 10 distinct blocks with higher values can still be seen along the diagonal. However, as perplexity increases, the rise in same-class similarity is not as significant as the rise in different-class similarity, making it difficult to distinguish between them.

3 Observation & Discussion

1) The impact of number of neighbors in KNN

From the experimental results, it can be observed that the number of neighbors (k)

	kernel	k=1	k=5	k=9	k=13
PCA	None	83.33% (25/30)	90.00% (27/30)	86.67% (26/30)	86.67% (26/30)
	Linear	80.00% (24/30)	83.33% (25/30)	83.33% (25/30)	80.00% (24/30)
	RBF	83.33% (25/30)	76.67% (23/30)	76.67% (23/30)	76.67% (23/30)
LDA	None	83.33% (25/30)	93.33% (28/30)	90.00% (27/30)	86.67% (26/30)
	Linear	80.00% (24/30)	80.00% (24/30)	76.67% (23/30)	80.00% (24/30)
	RBF	80.00% (24/30)	80.00% (24/30)	70.00% (21/30)	70.00% (21/30)

has little impact on face recognition accuracy. However, when k=13, the performance seems to be worse, as each subject has only 9 instances in the training set. When k>9, the neighbors will inevitably include instances from non-ground truth classes, which can cause confusion. On the other hand, for the RBF kernel, both PCA and LDA show the best performance when k=1. This suggests that when the data is projected into the feature space using the kernel and then dimensionality reduction is applied, the closest neighbors are typically from the same class.

2) The impact of γ of RBF kernel

	$\gamma = 1 \times 10^{-6}$	$\gamma = 1 \times 10^{-7}$	$\gamma = 1 \times 10^{-8}$	$\gamma = 1 \times 10^{-10}$
RBF PCA	40.00% (12/30)	76.67% (23/30)	76.67% (23/30)	80.00% (24/30)
RBF LDA	23.33% (7/30)	80.00% (24/30)	80.00% (24/30)	76.67% (23/30)

From the KNN experiment with $k=5$, it can be observed that when the γ is too large ($= 1 \times 10^{-6}$), the model's performance is poor. This may be because when the γ is large, the influence range of each point becomes smaller, meaning each data point has a larger impact on the decision boundary, which can lead to overfitting.

3) The impact of perplexity on SNE convergence

Perplexity	5	15	30	50	100
Converge iterations	750	250	300	200	200

From the results, it can be seen that a smaller perplexity requires more iterations for SNE to converge. This may be because a smaller perplexity value limits the range of neighbors for each point, causing the SNE to need more iterations to accurately adjust the relationships between these neighbors.

Additionally, in t-SNE, there is no clear convergence time. Although the spread of the low-dimensional embedding becomes more stable as the iterations increase, the axis range continues to expand, with the points becoming more spread out over time.