

ML Homework 07: Kernel Eigenfaces and t-SNE

- Student ID: 309553002
- Name: 林育愷

ML Homework 07: Kernel Eigenfaces and t-SNE

[Code with Detailed Explanations](#)

[Prerequisites](#)

[I. Kernel Eigenfaces](#)

[PCA](#)

[LDA](#)

[Eigenfaces, Fisherfaces, and Reconstruction](#)

[Using k-NN for Face Recognition](#)

[Kernel Tricks](#)

[Kernel PCA](#)

[Kernel LDA](#)

[II. t-SNE](#)

[Symmetric SNE](#)

[Visualizing the Embedding of Both t-SNE and Symmetric SNE](#)

[Visualizing the Distribution of Pairwise Similarities](#)

[Playing with Different Perplexity Values](#)

[Experiments Settings, Results, and Discussion](#)

[I. Kernel Eigenfaces](#)

[Eigenfaces, Fisherfaces, and Reconstruction](#)

[Face Recognition](#)

[II. t-SNE](#)

[Visualization of t-SNE and Symmetric SNE](#)

[References](#)

Code with Detailed Explanations

Prerequisites

I use Python 3.6 for this implementation based on platform Ubuntu 18.04, with the following packages

- PIL,
- NumPy,
- SciPy `scipy.spatial.distance`, and
- Matplotlib.

I. Kernel Eigenfaces

Firstly we load the Yale dataset, reading each file and store ones content as well as label as NumPy arrays (with downsampling).

```
1 from collections import namedtuple
2
```

```

3 yale = namedtuple('yale', ['data', 'label', 'image_shape'])
4 SHAPE = (50, 50)
5
6 def _load_data(directory, number_of_subjects):
7     data = []
8     label = []
9     for idx in range(1, number_of_subjects + 1):
10         pattern = 'subject%02d*.pgm' % idx
11         path = osp.join(directory, pattern)
12         filenames = glob.glob(path)
13         filenames.sort()
14         for filename in filenames:
15             label.append(idx)
16             image = Image.open(filename)
17             image = np.array(image.resize(SHAPE, Image.ANTIALIAS))
18             image = image.astype(float) / 255
19             data.append(image)
20
21     data = np.array(data)
22     label = np.array(label)
23
24     image_shape = data.shape[1:]
25     data = data.reshape((data.shape[0], -1))
26
27     dataset = yale(data, label, image_shape)
28
29     return dataset
30
31
32 def load_data(database_root='./Yale_Face_Database',
33               training_dir='Training',
34               testing_dir='Testing',
35               number_of_subjects=15):
36     training_dataset = _load_data(osp.join(database_root, training_dir),
37                                     number_of_subjects)
38     testing_dataset = _load_data(osp.join(database_root, testing_dir),
39                                  number_of_subjects)
40
41     return training_dataset, testing_dataset
42
43
44 training, testing = load_data()

```

PCA

Suppose that the training data matrix D is of shape $n \times d$, where n is the number of data points and d is the dimension of the data point. In our case we view an image as a large vector, so $d = 2500$, and $n = 135$.

We centralize the original data as D_c in the beginning, then we calculate eigenvalues and eigenvectors of the matrix $D_c^T D_c \in \mathbb{R}^{d \times d}$. Note that for our case, it takes some of time calculating such values. So we go back to the view of SVD

$$D_c = U \Sigma V^T.$$

What values we want are V and Σ , but in this case is harder than calculating U . A trick is that we obtain U by calculating eigenvectors of $D_c D_c^T$ at first. Later we calculate unnormalized V with

$$\begin{aligned} V\Sigma &= V\Sigma U^T U \\ &= D_c^T U. \end{aligned}$$

Since Σ is a diagonal matrix, we simply normalize each eigenvector of $D_c^T D_c$ (right singular vector of D_c) and then V is reconstructed.

We peak the eigenvectors of the top 25 largest eigenvalues as a basis of the feature space, and project each data to the feature space. We name the matrix of the 25 eigenvectors as

$V'_{\text{PCA}} \in \mathbb{R}^{d \times 25}$.

```

1  class PCA(object):
2      def __init__(self, data, dimension=25):
3          self.data = data
4          self.mean = data.mean(axis=0)
5          self.dimension = dimension
6          self.singular_values = np.array([])
7          self.left_singular_vectors = np.array([])
8          self.right_singular_vectors = np.array([])
9
10     @property
11     def feature_vectors(self):
12         return self.right_singular_vectors[:, :self.dimension]
13
14     @property
15     def feature_coordinates(self):
16         return self.dimensionality_reduction(self.data)
17
18     def dimensionality_reduction(self, data):
19         return (data - self.mean) @ self.feature_vectors
20
21     def compute(self):
22         matrix = self.data - self.mean
23         cov = matrix @ matrix.T
24
25         eigvals, eigvecs = np.linalg.eig(cov)
26         eigvals = np.where(eigvals < 0, 0, eigvals)
27
28         indices = np.argsort(eigvals)[::-1]
29         eigvals = eigvals[indices]
30         eigvecs = eigvecs[:, indices]
31
32         self.singular_values = np.sqrt(eigvals)
33         self.left_singular_vectors = eigvecs
34         self.right_singular_vectors = self.data.T @ eigvecs
35
36         for i in range(self.right_singular_vectors.shape[1]):
37             vector = self.right_singular_vectors[:, i]
38             vector = vector / np.linalg.norm(vector)
39             self.right_singular_vectors[:, i] = vector
40
41
42  pca = PCA(training.data)
43  pca.compute()

```

LDA

Suppose that we have data points with c classes $\mathcal{C}_1, \dots, \mathcal{C}_c$. The goal of LDA is to find the argument maximum of the object function of between-classes scatter S_B to within-class scatter S_W ,

$$\arg \max_{\|w\|=1} J(w) = \arg \max_{\|w\|=1} \frac{w^T S_B w}{w^T S_W w},$$

with

$$S_B = \frac{1}{|\mathcal{C}_j|} \sum_{j=1}^c (m_j - m)(m_j - m)^T,$$
$$S_W = \sum_{j=1}^c \sum_{i \in \mathcal{C}_j} (x_i - m_j)(x_i - m_j)^T,$$

where $|\mathcal{C}_j|$ is the number of data points within the class \mathcal{C}_j , $m = (\sum_{i=1}^n x_i)/n$, and $m_j = (\sum_{i \in \mathcal{C}_j} x_i)/|\mathcal{C}_j|$.

Using Rayleigh quotient, we become to calculate the eigenvalues and eigenvectors of the matrix $S_W^{-1} S_B$.

In our case S_W is not invertible, so we use the pseudo-inverse of S_W instead.

Again, we peak the eigenvectors of the top 25 largest eigenvalues as a basis of the feature space, and project each data to the feature space. We name the matrix of the 25 eigenvectors as $V'_{\text{LDA}} \in \mathbb{R}^{d \times 25}$.

```
1 class LDA(object):
2     def __init__(self, data, label, dimension=25):
3         self.data = data
4         self.mean = data.mean(axis=0)
5         self.degree = data.shape[1]
6         self.label = label
7         self.dimension = dimension
8
9         self.sw = np.array([])
10        self.sb = np.array([])
11
12        self.eigenvalues = np.array([])
13        self.eigenvectors = np.array([])
14
15    @property
16    def feature_vectors(self):
17        return self.eigenvectors[:, :self.dimension]
18
19    @property
20    def feature_coordinates(self):
21        return self.dimensionality_reduction(self.data)
22
23    def dimensionality_reduction(self, data):
24        return (data - self.mean) @ self.feature_vectors
25
26    def compute(self):
27        self.sw = np.zeros((self.degree, self.degree), dtype=float)
28        self.sb = np.zeros((self.degree, self.degree), dtype=float)
29
30        unique_labels = np.unique(self.label)
```

```

31         for label in unique_labels:
32             indices = np.argwhere(self.label == label).flatten()
33             data = self.data[indices]
34             mu = np.mean(data, axis=0)
35             self.sw += (data - mu).T @ (data - mu)
36             diff = np.array([mu - self.mean]).T
37             self.sb += len(indices) * diff @ diff.T
38         eigvals, eigvecs = np.linalg.eig(np.linalg.pinv(self.sw) @ self.sb)
39
40         # normalize eigenvectors
41         eigvecs = eigvecs.real
42         for idx in range(eigvecs.shape[1]):
43             eigvecs[:, idx] /= np.linalg.norm(eigvecs[:, idx])
44
45         # sort eigenvalues and eigenvectors
46         indices = np.argsort(eigvals)[::-1]
47
48         self.eigenvalues = eigvals[indices]
49         self.eigenvectors = eigvecs[:, indices]
50
51
52     lda = LDA(training.data, training.label)
53     lda.compute()

```

Eigenfaces, Fisherfaces, and Reconstruction

For visualizing eigenfaces (PCA) and fisherfaces (LDA), we simply reshape each eigenvector of V'_{PCA} and V'_{LDA} as the original shape of image respectively. For reconstruction of a given coordinates v of the feature space of PCA or LDA, we reconstruct ones image I by calculating

$$I_* = V'_*v + m,$$

where the symbol $*$ would be PCA or LDA, and m is the mean of the training data.

```

1  _, axs = plt.subplots(2, 5)
2
3  indices = np.random.choice(training.data.shape[0], size=10)
4  feature_coordinates = pca.feature_coordinates[indices]
5
6  for i in range(2):
7      for j in range(5):
8          idx = i * 5 + j
9          image = pca.feature_vectors @ feature_coordinates[idx] + pca.mean
10         image = image.reshape(training.image_shape)
11         axs[i, j].imshow(image, cmap='gray')
12
13     fig, axs = plt.subplots(5, 5)
14     for i in range(5):
15         for j in range(5):
16             idx = i * 5 + j
17             image = lda.feature_vectors[:, idx]
18             image = image.reshape(training.image_shape)
19             axs[i, j].imshow(image, cmap='gray')

```

Using k-NN for Face Recognition

We use data in feature space given by PCA or LDA to recognize faces using k-NN algorithm. In practice, we find the top k nearest training data points in feature space for an input data, and determine the class by voting.

```
1 def knn(source, label, targets, k=7):
2     result = []
3     for target in targets:
4         distance = np.linalg.norm(source - target, axis=1)
5         indices = np.argsort(distance)[:k]
6         labels = label[indices]
7         voting = np.argmax(np.bincount(labels))
8         result.append(voting)
9
10    return np.array(result)
11
12
13 pca_coordinates = pca.dimensionality_reduction(testing.data)
14 for k in range(3, 16, 2):
15     pca_result = knn(pca.feature_coordinates, training.label,
16                       pca_coordinates, k)
17     acc = np.where(pca_result == testing.label, 1, 0).mean() * 100
18     print('PCA with k=%02d -- Accuracy: %.2f%%' % (k, acc))
19
20 lda_coordinates = lda.dimensionality_reduction(testing.data)
21 for k in range(3, 16, 2):
22     lda_result = knn(lda.feature_coordinates, training.label,
23                       lda_coordinates, k)
24     acc = np.where(lda_result == testing.label, 1, 0).mean() * 100
25     print('LDA with k=%02d -- Accuracy: %.2f%%' % (k, acc))
```

Kernel Tricks

Given the gram (kernel) matrix $K = [k(x_i, x_j)]_{i,j} \in \mathbb{R}^{n \times n}$ based on linear kernel, polynomial kernel, or RBF kernel,

$$\begin{aligned} k_{\text{linear}}(x_i, x_j) &= \langle x_i, x_j \rangle, \\ k_{\text{polynomial}}(x_i, x_j) &= (\langle x_i, x_j \rangle + c)^d, \\ k_{\text{RBF}}(x_i, x_j) &= \exp(-\gamma \|x_i - x_j\|_2^2), \end{aligned}$$

we are going to do the dimensionality reduction on the feature space provided by kernel function k .

```
1 def linear_kernel(x, y=None):
2     y = x if y is None else y
3     return cdist(x, y, lambda u, v: u @ v)
4
5
6 def polynomial_kernel(x, y=None, c=1, d=2):
7     y = x if y is None else y
8     return cdist(x, y, lambda u, v: (u @ v + c)**d)
9
10
11 def rbf_kernel(x, y=None, gamma=1e-3):
12     y = x if y is None else y
```

```

13     dist = cdist(x, y, 'sqeuclidean')
14     return np.exp(-gamma * dist)
15
16
17 class Kernel(object):
18     LINEAR = linear_kernel
19     POLYNOMIAL = polynomial_kernel
20     RBF = rbf_kernel

```

Kernel PCA

According to ¹, we centralize K by calculating $K' = K - 1_N K - K 1_N + 1_N K 1_N$, where $1_N = [1/N]_{i,j} \in \mathbb{R}^{N \times N}$. Then we obtain an orthonormal basis by solving eigenvalue problem of K' .

```

1 class KernelPCA(object):
2     def __init__(self, data, kernel_type, dimension=25):
3         self.data = data
4         self.dimension = dimension
5         self.eigenvalues = np.array([])
6         self.eigenvectors = np.array([])
7
8         self.kernel_function = kernel_type
9         self.kernel = self.kernel_function(self.data)
10        self.centralized_kernel = self.centralize(self.kernel)
11
12    def centralize(self, kernel):
13        # https://en.wikipedia.org/wiki/kernel_principal_component_analysis
14        ones = np.ones(kernel.shape) / kernel.shape[0]
15        return kernel - ones @ kernel - kernel @ ones + ones @ kernel @ ones
16
17    @property
18    def feature_vectors(self):
19        return self.eigenvectors[:, :self.dimension]
20
21    @property
22    def feature_coordinates(self):
23        return self.centralized_kernel @ self.feature_vectors
24
25    def compute(self):
26        eigvals, eigvecs = np.linalg.eig(self.centralized_kernel)
27        eigvals = np.where(eigvals < 0, 0, eigvals)
28
29        indices = np.argsort(eigvals)[::-1]
30        eigvals = eigvals[indices]
31        eigvecs = eigvecs[:, indices]
32
33        self.eigenvalues = eigvals
34        self.eigenvectors = eigvecs.real
35
36        for i in range(self.eigenvectors.shape[1]):
37            vector = self.eigenvectors[:, i]
38            vector = vector / np.linalg.norm(vector)
39            self.eigenvectors[:, i] = vector
40
41
42    ktypes = [Kernel.LINEAR, Kernel.POLYNOMIAL, Kernel.RBF]

```

```

43 knames = ['Linear', 'Polynomial', 'RBF']
44 for ktype, kname in zip(ktypes, knames):
45     pca = KernelPCA(dataset.data, ktype)
46     pca.compute()
47     pca_training = pca.feature_coordinates[:training.data.shape[0]]
48     pca_testing = pca.feature_coordinates[training.data.shape[0]:]
49     for k in range(3, 16, 2):
50         pca_result = knn(pca_training, training.label, pca_testing, k)
51         acc = np.where(pca_result == testing.label, 1, 0).mean() * 100
52         print('%s Kernel PCA with k=%02d -- Accuracy: %.2f%%' %
53               (kname, k, acc))
54         print('-' * 66)

```

Kernel LDA

According to ², solving LDA in kernel space is equivalent to solving

$$\arg \max_{\|w\|=1} J_k(w) = \arg \max_{\|\alpha\|=1} \frac{\alpha^T M \alpha}{\alpha^T N \alpha},$$

with

$$M = \sum_{j=1}^c |\mathcal{C}_j| (\bar{m}_j - \bar{m})(\bar{m}_j - \bar{m})^T,$$

$$N = \sum_{j=1}^c K_j (I - \mathbf{1}_{|\mathcal{C}_j|}) K_j^T,$$

where

$$\bar{m} = \left[\frac{1}{n} \sum_{j=1}^n k(x_i, x_j) \right]_i \in \mathbb{R}^n,$$

$$\bar{m}_j = \left[\frac{1}{|\mathcal{C}_j|} \sum_{j \in \mathcal{C}_j} k(x_i, x_j) \right]_i \in \mathbb{R}^n,$$

$$K_j = [k(x_i, x_l)]_{i=1, \dots, n; l \in \mathcal{C}_j} \in \mathbb{R}^{n \times |\mathcal{C}_j|}.$$

Similarly, we solve the problem by finding eigenvalues and eigenvectors of $N^{-1}M$.

```

1  class KernelLDA(object):
2      def __init__(self, data, label, kernel_type, dimension=25):
3          self.data = data
4          self.size = data.shape[0]
5          self.degree = data.shape[1]
6          self.label = label
7          self.dimension = dimension
8
9          self.m = np.array([])
10         self.n = np.array([])
11
12         self.eigenvalues = np.array([])
13         self.eigenvectors = np.array([])
14
15         self.kernel_function = kernel_type
16         self.kernel = self.kernel_function(self.data)
17         self.centralized_kernel = self.centralize(self.kernel)
18

```



```

19     def centralize(self, kernel):
20         # https://en.wikipedia.org/wiki/Kernel\_principal\_component\_analysis
21         ones = np.ones(kernel.shape) / kernel.shape[0]
22         return kernel - ones @ kernel - kernel @ ones + ones @ kernel @ ones
23
24     @property
25     def feature_vectors(self):
26         return self.eigenvectors[:, :self.dimension]
27
28     @property
29     def feature_coordinates(self):
30         return self.centralized_kernel @ self.feature_vectors
31
32     def compute(self):
33         self.m = np.zeros((self.size, self.size), dtype=float)
34         self.n = np.zeros((self.size, self.size), dtype=float)
35
36         unique_labels = np.unique(self.label)
37         mean = np.mean(self.kernel, axis=1)
38         for label in unique_labels:
39             indices = np.argwhere(self.label == label).flatten()
40             kernel = self.kernel[indices]
41             size = len(indices)
42             mean_label = np.mean(kernel, axis=0)
43
44             mdiff = mean_label - mean
45             self.m += size * np.kron(mdiff, mdiff).reshape((self.size, -1))
46             ones = np.ones((size, size)) / size
47             self.n += kernel.T @ (np.eye(size) - ones) @ kernel
48         eigvals, eigvecs = np.linalg.eig(np.linalg.pinv(self.m) @ self.n)
49
50         # normalize eigenvectors
51         eigvecs = eigvecs.real
52         for idx in range(eigvecs.shape[1]):
53             eigvecs[:, idx] /= np.linalg.norm(eigvecs[:, idx])
54
55         # sort eigenvalues and eigenvectors
56         indices = np.argsort(eigvals)[::-1]
57
58         self.eigenvalues = eigvals[indices]
59         self.eigenvectors = eigvecs[:, indices]
60
61
62 ktypes = [Kernel.LINEAR, Kernel.POLYNOMIAL, Kernel.RBF]
63 knames = ['Linear', 'Polynomial', 'RBF']
64 for ktype, kname in zip(ktypes, knames):
65     lda = KernelLDA(dataset.data, dataset.label, ktype)
66     lda.compute()
67     lda_training = lda.feature_coordinates[:training.data.shape[0]]
68     lda_testing = lda.feature_coordinates[training.data.shape[0]:]
69     for k in range(3, 16, 2):
70         lda_result = knn(lda_training, training.label, lda_testing, k)
71         acc = np.where(lda_result == testing.label, 1, 0).mean() * 100
72         print('%s Kernel LDA with k=%02d -- Accuracy: %.2f%%' %
73               (kname, k, acc))
74     print('-' * 66)

```

II. t-SNE

I use the source code of t-SNE written by Laurens van der Maaten ³, and try to modify it.

Symmetric SNE

Based on source code, there are two differences between t-SNE and symmetric SNE:

- t-SNE uses student-t distribution in low dimension but symmetric SNE uses normal distribution.

```
1 # t-SNE
2 # Compute pairwise affinities
3 sum_Y = np.sum(np.square(Y), 1)
4 num = -2. * np.dot(Y, Y.T)
5 num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
6 num[range(n), range(n)] = 0.
7 Q = num / np.sum(num)
8 Q = np.maximum(Q, 1e-12)
```

```
1 # symmetric SNE
2 # Compute pairwise affinities
3 num = np.exp(-cdist(Y, Y, 'sqeuclidean'))
4 num[range(n), range(n)] = 0.
5 Q = num / np.sum(num)
6 Q = np.maximum(Q, 1e-12)
```

- Partial derivative of C for t-SNE with respect to y_i is written by

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

but the one for symmetric SNE with respect to y_i is given by

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

```
1 # t-SNE
2 # Compute gradient
3 PQ = P - Q
4 for i in range(n):
5     dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T *
6                       (Y[i, :] - Y), 0)
```

```
1 # symmetric SNE
2 # Compute gradient
3 PQ = P - Q
4 for i in range(n):
5     dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y),
6                       0)
```

Visualizing the Embedding of Both t-SNE and Symmetric SNE

I `yield` embedding Y in each iteration and plot them using `matplotlib`.

```
1 def symmetric_sne(X=np.array([]), no_dims=2, initial_dims=50,
2   perplexity=30.0):
3     # Check inputs
4     # ...
5     # Initialize variables
6     # ...
7
8     # Run iterations
9     for iter in range(max_iter):
10         # Compute pairwise affinities
11         # Compute gradient
12         # Perform the update
13         # ...
14
15         # Return solution for each iteration
16         yield Y, P, Q
17
18
19 def tsne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0):
20     # Check inputs
21     # ...
22
23     # Initialize variables
24     # ...
25
26     # Run iterations
27     for iter in range(max_iter):
28         # Compute pairwise affinities
29         # Compute gradient
30         # Perform the update
31         # ...
32
33         # Return solution for each iteration
34         yield Y, P, Q
35
36
37 x = np.loadtxt("mnist2500_X.txt")
38 labels = np.loadtxt("mnist2500_labels.txt")
39 methods = [tsne, symmetric_sne]
40 method_names = ['tsne', 'ssne']
41 perplexity = 20
42
43 for method, name in zip(methods, method_names):
44     title = '%s_%d' % (name, perplexity)
45     fig = plt.figure()
46     frames = []
47     Y = np.array([])
48     for idx, (Y, P, Q) in enumerate(method(X, 2, 50, 5)):
49         if idx % 3 == 0: # downsampling
50             frames.append((plt.scatter(Y[:, 0], Y[:, 1], 20, labels), ))
51
52
```

```

53     interval = int(10000 / len(frames))
54     ani = animation.ArtistAnimation(fig,
55                                     frames,
56                                     interval=interval,
57                                     repeat=False)
58     ani.save('images/%s.gif' % title, writer='pillow')

```

Visualizing the Distribution of Pairwise Similarities

I return P and Q in both functions `tsne` and `symmetric_sne`, reorder data with labels, and save them as image with colormap.

```

1  for method, name in zip(methods, method_names):
2      title = '%s_%d' % (name, perplexity)
3      P = np.array([])
4      Q = np.array([])
5      for idx, (Y, P, Q) in enumerate(method(X, 2, 50, 5)):
6          # ...
7
8          # reordering
9          indices = np.argsort(labels).flatten()
10         P = P[indices]
11         P = P[:, indices]
12         # setting maximum color range as
13         # [np.min(P), np.quantile(P, 0.985)]
14         P[P >= np.quantile(P, 0.985)] = np.quantile(P, 0.985)
15         fig = plt.figure('High Dimension Pairwise Similarity')
16         plt.imsave('images/%s_hd.png' % title, P, cmap='hot')
17
18         # reordering
19         Q = Q[indices]
20         Q = Q[:, indices]
21         # setting maximum color range as
22         # [np.min(P), np.quantile(P, 0.985)]
23         Q[Q >= np.quantile(Q, 0.985)] = np.quantile(Q, 0.985)
24         fig = plt.figure('Low Dimension Pairwise Similarity')
25         plt.imsave('images/%s_ld.png' % title, Q, cmap='hot')

```

Playing with Different Perplexity Values

I write a script to run various tasks of different perplexity values. Values can be passed from script to main code using `argparser`.

```

1  # script
2  import multiprocessing as mp
3  import subprocess as sp
4
5  N_CPUS = mp.cpu_count()
6
7
8  def task(perplexity):
9      command = ' '.join([
10          'python3', 'HW07_2_tsNE.py',
11          str(perplexity), '--enable-part-1', '--enable-part-2',
12          '--enable-part-3'
13      ])

```

```

14     sp.check_output(command, shell=True)
15
16
17     if __name__ == '__main__':
18         with mp.Pool(N_CPUS) as pool:
19             pool.map(task, [5, 10, 15, 20, 25, 30])

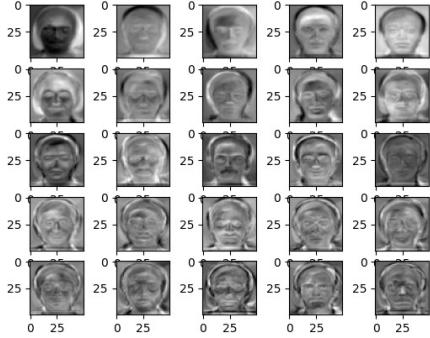
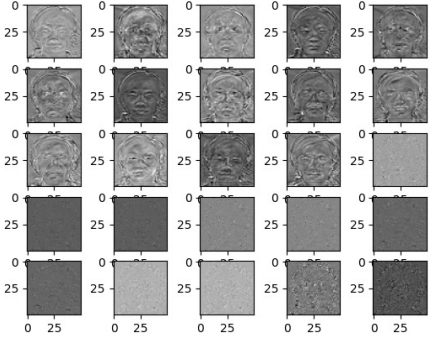
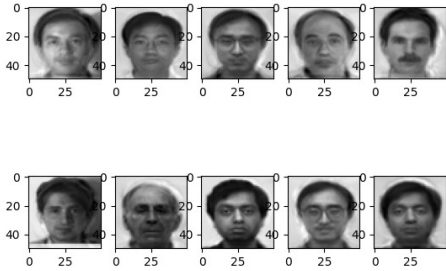
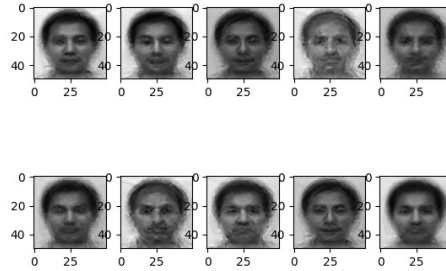
```

Experiments Settings, Results, and Discussion

I. Kernel Eigenfaces

Eigenfaces, Fisherfaces, and Reconstruction

We can see that eigenfaces visually have distinct appearances, and it has good reconstruction of eigenfaces. On the other hand, we can see the contour in the only first fisherface since it is the argument that can maximize the ratio J defined in LDA. Hence fisherfaces are not good for reconstruction.

Eigenfaces	Fisherfaces
	
Reconstruction of eigenfaces	Reconstruction of fisherfaces
	

Face Recognition

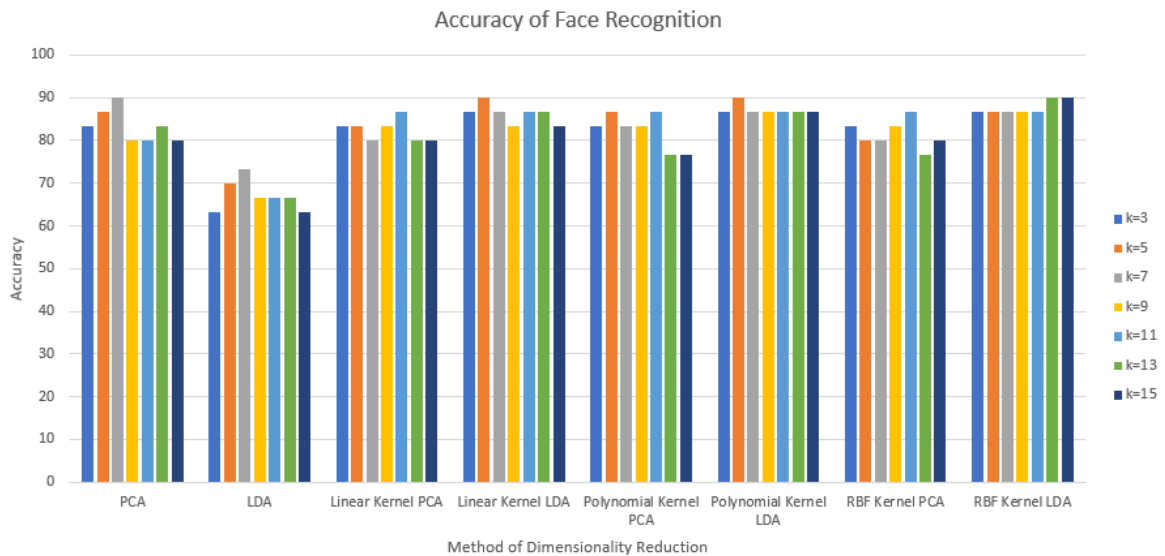
Following is the result of face recognition using PCA, LDA, and kernel tricks of them. Here observes that

- With well parameters of kernel, overall accuracy of LDA kernel tricks is better than LDA, but PCA is better than any of its kernel tricks.
- Accuracy of PCA is better than LDA, but in kernel tricks, overall kernel LDAs are better than kernel PCAs.

- With original algorithm for Yale dataset, computational costs of PCA and LDA (eigenvalue problem of $d \times d$ matrix) are heavier than the one their kernel tricks take (eigenvalue problem of $n \times n$ matrix). Since we modify the algorithm of PCA, the cost is reduced near the one kernel tricks of PCA take.

Table 1. Accuracy of face recognition using various methods (unit: %)

k	PCA	LDA	Linear Kernel PCA	Linear Kernel LDA	Polynomial Kernel PCA ($c, d) = (1, 2)$	Polynomial Kernel LDA ($c, d) = (1, 2)$	RBF Kernel PCA $\gamma = 10^{-3}$	RBF Kernel LDA $\gamma = 10^{-3}$
3	83.33	96.67	83.33	86.67	83.33	86.67	83.33	86.67
5	86.67	96.67	83.33	90	86.67	90	80	86.67
7	90	96.67	80	86.67	83.33	86.67	80	86.67
9	80	96.67	83.33	83.33	83.33	86.67	83.33	86.67
11	80	96.67	86.67	86.67	86.67	86.67	86.67	86.67
13	83.33	96.67	80	86.67	76.67	86.67	76.67	90
15	80	96.67	80	83.33	76.67	86.67	80	90



II. t-SNE

Visualization of t-SNE and Symmetric SNE

In table 2, we can see that

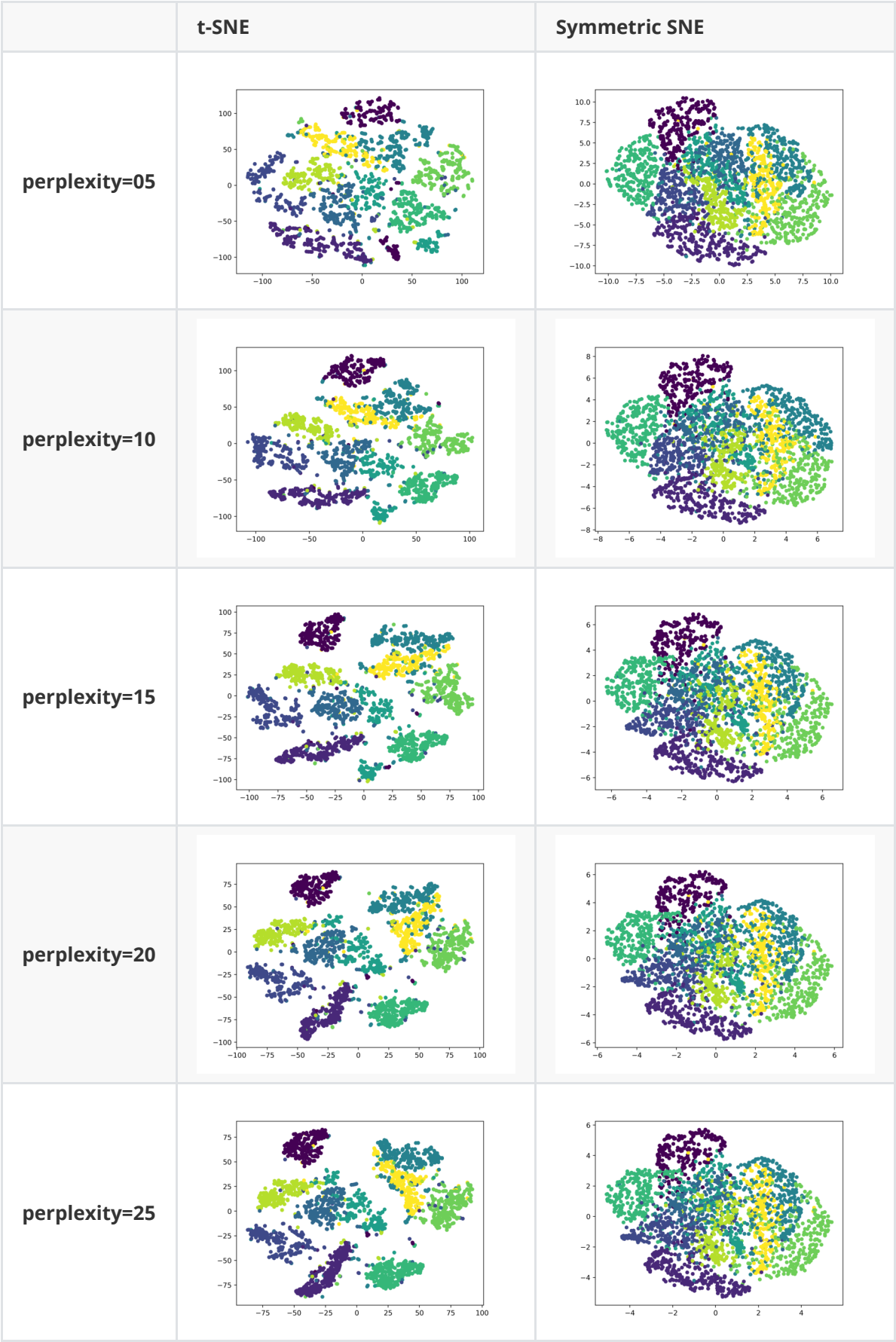
- Symmetric SNE has the crowded issue comparing to t-SNE, no matter what value of perplexity is.
- With larger perplexity value, we can see that in t-SNE, each class is more clustered, but distance of each two classes is smaller (be ware of the scale of each image).
- We cannot see meaningful observation in symmetric SNE with different perplexity values.

In table 3, we observe that

- With ordering, we can see totally 10 bright blocks on the diagonal of each matrix of pairwise similarities.

Table 2. Embedding of t-SNE and Symmetric SNE

You can click each image to see the GIF procedure of iteration.



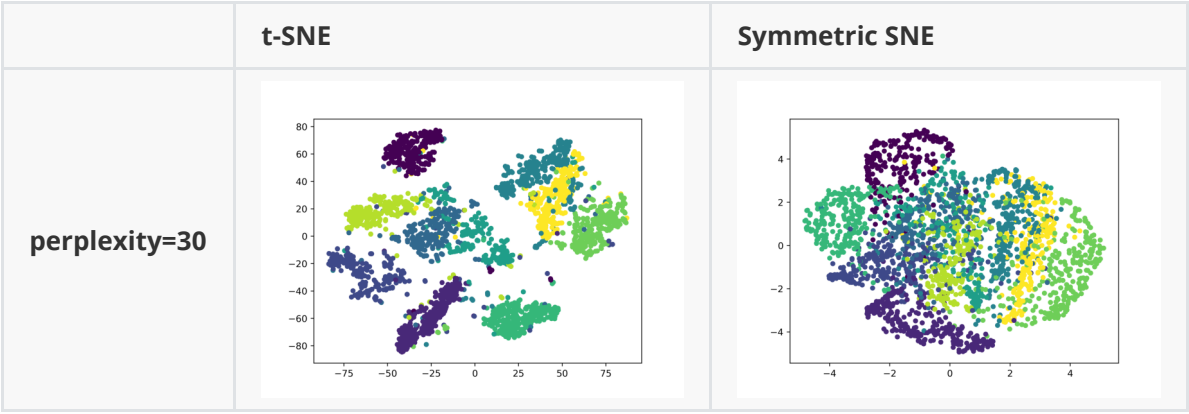
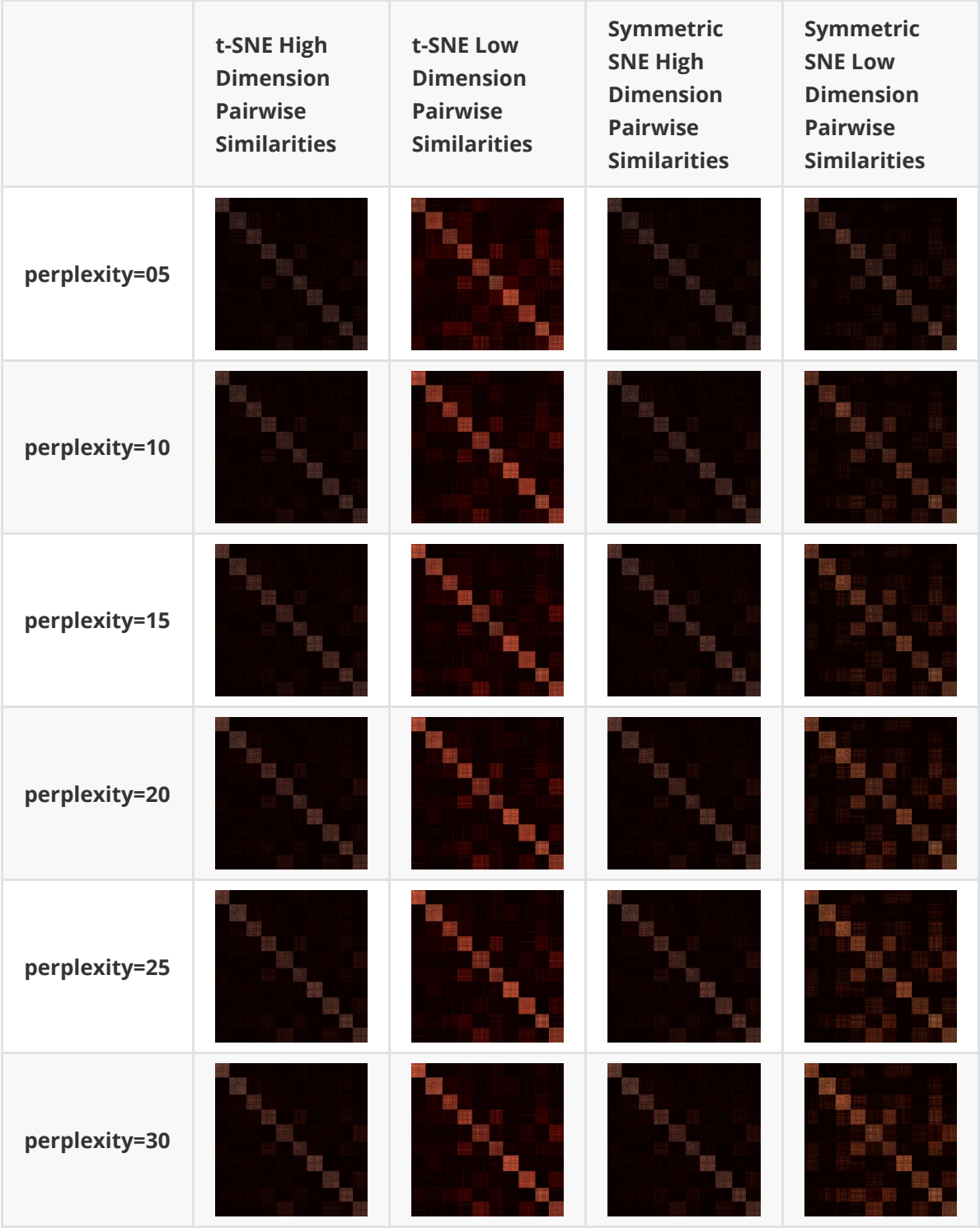


Table 3. Visualization of distribution of pairwise similarities



References

-
1. https://en.wikipedia.org/wiki/Kernel_principal_component_analysis. ↗
 2. https://en.wikipedia.org/wiki/Kernel_Fisher_discriminant_analysis ↗
 3. <https://lvdmaaten.github.io/tsne/> ↗