

Machine Learning Homework 7

Machine Learning Homework 7

Environment

Code with detailed explanations

Kernel Eigenfaces

Part1

Part2

Part3

t-SNE

Part1

Part2

Part3

Part4

Experiments settings and results & Discussion

Kernel Eigenfaces

Part1

Part2

Part3

Observations

Results

t-SNE

Part1

Observations

Results

Part2

Part3

Part4

Observations

Results

Observations and discussion

Meaning of eigenface

Environment

- Language: Python
- Version: 3.9.16

Code with detailed explanations

Kernel Eigenfaces

Part1

- Algorithms
 - PCA
1. Calculate covariance matrix of data X (D by D matrix)

$$S = \left[\frac{1}{N} \sum_{\mathbf{x}} (\mathbf{x} - \bar{\mathbf{x}})(\mathbf{x} - \bar{\mathbf{x}})^T \right]$$

2. According to *Rayleigh quotient*,

we could use eigen decomposition on covariance matrix of data to get the first k largest eigenvectors (principal components) as W orthogonal projection matrix

$$\mathbf{W}^\top S \mathbf{W}$$

3. Project the data onto low dimensional space

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

4. Reconstruct the data (lossless if # of eigenvectors = D)

$$\mathbf{X} = \mathbf{Z}\mathbf{W}^T$$

- LDA

1. Within-class scatter S_W

$$\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in C_j} \mathbf{x}_i$$

$$S_W = \sum_{j=1}^k \sum_{i \in C_j} (\mathbf{x}_i - \mathbf{m}_j)(\mathbf{x}_i - \mathbf{m}_j)^T$$

2. Between-class scatter S_B

$$\mathbf{m} = \frac{1}{n} \sum \mathbf{x}$$

$$S_B = \sum_{j=1}^k (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^T$$

3. According to the objective function $J(W) = \frac{\det(W^T S_B W)}{\det(W^T S_W W)}$ and *Rayleigh quotient*,

we could formulate it to

$$S_B w_l = \lambda_l S_W w_l$$

$$S_W^{-1} S_B w_l = \lambda_l w_l$$

4. Use eigen decomposition on $S_W^{-1} S_B$ to get the first k largest eigenvectors (principal components) as W orthogonal projection matrix

5. Project the data onto low dimensional space

$$\mathbf{Z} = \mathbf{X}\mathbf{W}$$

6. Reconstruct the data (lossless if # of eigenvectors = D)

$$\mathbf{X} = \mathbf{Z}\mathbf{W}^T$$

- `utils.load_data` function: Load data from PGM files

Arguments

files: list of files, contains PGM files

Steps

1. Load the image
2. Center crop image to squared shape
3. Resize the cropped image to 40 by 40 (need less computation resources)

4. Scale it from [0, 255] to [0, 1]

```
14 def load_data(
15     files: list[str],
16 ) -> tuple[npt.NDArray[np.float64], npt.NDArray[np.int32], tuple[int, int]]:
17     CROPED_IMAGE_SIZE = (40, 40)
18     files.sort()
19
20     text = " ".join(files)
21     number_of_images_per_subjects = text.count("subject01")
22     number_of_subjects = len(files) // number_of_images_per_subjects
23
24     labels = np.repeat(list(range(number_of_subjects)), number_of_images_per_subjects)
25
26     images = []
27     for file in files:
28         image = Image.open(file)
29         width, height = image.size
30
31         # center crop
32         short_side = min(width, height)
33         left = (width - short_side) / 2
34         top = (height - short_side) / 2
35         right = width - left
36         bottom = height - top
37         image = image.crop((round(left), round(top), round(right), round(bottom)))
38
39         # resize to 40 x 40
40         image = image.resize(CROPED_IMAGE_SIZE, resample=Image.Resampling.BILINEAR)
41
42         images.append(np.array(image, dtype=np.float64).flatten() / 255.0)
43         image.close()
44
45     images = np.stack(images, axis=0)
46     return images, labels, CROPED_IMAGE_SIZE
```

- `train.py` main: program entry point

In Part1, we only focus on normal PCA and LDA.

Here, the covariance matrix calculation (`np.cov`) corresponds to the first step of PCA.

PCA: Calculate covariance matrix of data X (D by D matrix)

$$S = \left[\frac{1}{N} \sum_x (x - \bar{x})(x - \bar{x})^\top \right]$$

```

149 if __name__ == "__main__":
150     args = Arguments().parse_args()
151
152     os.makedirs(args.out_dir, exist_ok=True)
153
154     pattern = os.path.join(args.root_dir, TRAIN_FOLDER, "*.pgm")
155     files = glob.glob(pattern)
156
157     data, labels, _ = load_data(files)
158
159     if args.kernel != "none":
160         # PCA assumes the kernel is centered
161         centered_x = kernel(data, data, args.kernel, center=True) / data.shape[0]
162         x = kernel(data, data, args.kernel)
163
164         pca_weights = solve_by_pca(centered_x, args.kernel, args.num_components)
165         fisher_weights = solve_by_fisher(x, labels, args.kernel, args.num_components)
166         np.save(os.path.join(args.out_dir, "kernel_eigen.npy"), pca_weights)
167         np.save(os.path.join(args.out_dir, "kernel_fisher.npy"), fisher_weights)
168     else:
169         x = np.cov(data, rowvar=False, bias=True)
170
171         pca_weights = solve_by_pca(x, args.kernel, args.num_components)
172         fisher_weights = solve_by_fisher(data, labels, args.kernel, args.num_components)
173         np.save(os.path.join(args.out_dir, "eigen.npy"), pca_weights)
174         np.save(os.path.join(args.out_dir, "fisher.npy"), fisher_weights)
175

```

- `train.solve_by_pca` function: use PCA algorithm to find W orthogonal projection matrix

Arguments

`x`: np.ndarray, covariance matrix of data X
`kernel_type`: ignored
`num_components`: keep first k principal components

In Part1, we only focus on normal PCA.

Here, it is the second step of PCA.

According to *Rayleigh quotient*,
 we could use eigen decomposition on covariance matrix of data to get the first k largest eigenvectors (principal components) as W orthogonal projection matrix

$$W^T S W$$

```

27 def solve_by_pca(
28     x: npt.NDArray[np.float64], kernel_type: KERNEL_TYPES, num_components: int
29 ) -> npt.NDArray[np.float64]:
30     # x: D by D matrix (normal)
31     # x: N by N matrix (kernel)
32
33     if num_components > x.shape[0]:
34         text = "the number of features"
35         if kernel_type != "none":
36             text = "the number of samples"
37         raise ValueError(f"The num_components should not be larger than {text}.")
38
39     start_time = timeit.default_timer()
40
41     eigenvalues, eigenvectors = np.linalg.eigh(x)
42
43     print(
44         "Finished solving eigenvectors in",
45         timeit.default_timer() - start_time,
46         "seconds",
47     )
48
49     # find num_components largest eigenvectors
50     indices = np.argsort(eigenvalues)[-num_components - 1 : -1]
51     eigenvectors = eigenvectors[:, indices]
52     return eigenvectors

```

- `train.calculate_between_and_with_class_covariance` function: calculate between-class and within-class covariance

Arguments

`x`: np.ndarray, features of data X
`labels`: np.ndarray, gt labels of data X
`kernel_type`: ignored
`num_components`: keep first k principal components

In Part1, we only focus on normal LDA.

Here, it corresponds to the first and second step of LDA.

1. Within-class scatter S_W

$$\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in C_j} \mathbf{x}_i$$

$$S_W = \sum_{j=1}^k \sum_{i \in C_j} (\mathbf{x}_i - \mathbf{m}_j)(\mathbf{x}_i - \mathbf{m}_j)^T$$

2. Between-class scatter S_B

$$\mathbf{m} = \frac{1}{n} \sum \mathbf{x}$$

$$S_B = \sum_{j=1}^k (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^T$$

```

55     def calculate_between_and_with_class_covariance(
56         x: npt.NDArray[np.float64], labels: npt.NDArray[np.int32], kernel_type: KERNEL_TYPES
57     ) -> tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]]:
58         # calculate global mean: normal + kernel
59         global_mean: npt.NDArray[np.float64] = np.mean(x, axis=0, keepdims=True)
60
61         # calculate with-class & between-class covariance
62         groups, counts = np.unique(labels, return_counts=True)
63         covariance_between_class = np.zeros((x.shape[1],) * 2, dtype=np.float64)
64         covariance_within_class = np.zeros_like(covariance_between_class)
65         for i in groups:
66             group_x = x[labels == i]
67             group_mean: npt.NDArray[np.float64] = np.mean(group_x, axis=0, keepdims=True)
68
69             # between-class covariance: normal + kernel
70             centered_group_mean = group_mean - global_mean
71             covariance_between_class += (
72                 counts[i] * centered_group_mean.T @ centered_group_mean
73             )
74
75         if kernel_type == "none":
76             # within-class covariance: normal
77             centered_group_x = group_x - group_mean
78             covariance_within_class += np.dot(centered_group_x.T, centered_group_x)
79         else:
80             # within-class covariance: kernel
81             group_identity = np.identity(counts[i])
82             covariance_within_class += (
83                 group_x.T
84                 @ (group_identity - np.full_like(group_identity, 1 / counts[i]))
85                 @ group_x
86             )
87
88         return covariance_between_class, covariance_within_class

```

- `train.solve_by_fisher` function: use LDA algorithm to find W orthogonal projection matrix

Arguments

x : np.ndarray, covariance matrix of data X
 $labels$: np.ndarray, gt labels of data X
 $kernel_type$: ignored
 $num_components$: keep first k principal components

In Part1, we only focus on normal LDA.

After calculating the between-class and within-class covariance,

use eigen decomposition on $S_W^{-1}S_B$ to get the first k largest eigenvectors (principal components) as W orthogonal projection matrix.

$$S_W^{-1}S_Bw_l = \lambda_l w_l$$

```

91     def solve_by_fisher(
92         x: npt.NDArray[np.float64],
93         labels: npt.NDArray[np.int32],
94         kernel_type: KERNEL_TYPES,
95         num_components: int,
96     ) → npt.NDArray[np.float64]:
97         # x: N by D matrix
98         # labels: N array
99
100        if num_components > x.shape[1]:
101            text = "the number of features"
102            if kernel_type ≠ "none":
103                text = "the number of samples"
104            raise ValueError(f"The num_components should not be larger than {text}.")
105
106        start_time = timeit.default_timer()
107        (
108            covariance_between_class,
109            covariance_within_class,
110        ) = calculate_between_and_with_class_covariance(x, labels, kernel_type)
111
112        print(
113            "Finished calculating covariances in",
114            timeit.default_timer() - start_time,
115            "seconds",
116        )
117
118    start_time = timeit.default_timer()
119
120    #  $S_w^{-1} * S_b$ 
121    inverted_covariance_within_class = np.linalg.pinv(covariance_within_class)
122    eigenvalues, eigenvectors = np.linalg.eigh(
123        inverted_covariance_within_class @ covariance_between_class
124    )
125
126    print(
127        "Finished solving eigenvectors in",
128        timeit.default_timer() - start_time,
129        "seconds",
130    )
131
132    # find num_components largest eigenvectors
133    indices = np.argsort(eigenvalues)[:-num_components - 1:-1]
134    eigenvectors = eigenvectors[:, indices]
135    return eigenvectors

```

- `test.py` main: program entry point

```

192 ✓ if __name__ == "__main__":
193     args = Arguments().parse_args()
194
195     # load data
196     train_data, train_labels, image_size = load_dataset(args.root_dir, TRAIN_FOLDER)
197     test_data, test_labels, _ = load_dataset(args.root_dir, TEST_FOLDER)
198
199     # load weights
200     pca_weights = np.load(args.eigen_path)
201     fisher_weights = np.load(args.fisher_path)
202
203     if args.kernel == "none":
204         visualize_faces(
205             args.out_dir,
206             train_data,
207             pca_weights,
208             image_size,
209             num_eigenfaces=args.num_eigenfaces,
210             num_reconstructed_faces=args.num_reconstructed_faces,
211             file_name="eigenfaces.jpg",
212         )
213     else:
214         visualize_faces(
215             args.out_dir,
216             train_data,
217             fisher_weights,
218             image_size,
219             num_eigenfaces=args.num_eigenfaces,
220             num_reconstructed_faces=args.num_reconstructed_faces,
221             file_name="fisherfaces.jpg",
222         )
223     evaluate(
224         train_data,
225         train_labels,
226         test_data,
227         test_labels,
228         pca_weights,
229         fisher_weights,
230         args.k_neighbors,
231     )
232 else:
233     evaluate_kernel(
234         train_data,
235         train_labels,
236         test_data,
237         test_labels,
238         pca_weights,
239         fisher_weights,
240         args.k_neighbors,
241     )
242

```

- `utils.plot_faces` function: plot faces on image grid and export image

Arguments

`file_name`: str, image file name
`samples`: np.ndarray, faces
`cols_per_row`: int, used for plot image grid

```

77 def plot_faces(file_name: str, samples: npt.NDArray[np.float64], cols_per_row: int):
78     fig = plt.figure()
79     grid = ImageGrid(
80         fig,
81         111,
82         nrows_ncols=(math.ceil(samples.shape[0] / cols_per_row), cols_per_row),
83         axes_pad=0,
84     )
85
86     for i, ax in enumerate(grid):
87         ax: plt.Axes
88         ax.imshow(samples[i], cmap="gray")
89         ax.xaxis.set_visible(False)
90         ax.yaxis.set_visible(False)
91
92     fig.savefig(file_name)
93     plt.close(fig)

```

- `test.visualize_faces` function: visualize eigenfaces, fisherfaces and reconstructed faces

Arguments

out_dir: str, output folder path
 samples: np.ndarray, shape (N, D), data samples
 weights: PCA or LDA weights, shape (D, E)
 image_size: tuple[int, int], (width, height), data image size
 file_name: str, base output file name
 num_eigenfaces: int, how many eigenfaces need to be visualized
 num_reconstructed_faces: int, how many faces need to be reconstructed
 cols_per_row: int, used for plot image grid

After plot the eigenfaces,

we reconstruct faces by the formula $X = XWW^T$.

```

60  def visualize_faces(
61      out_dir: str,
62      samples: npt.NDArray[np.float64],
63      weights: npt.NDArray[np.float64],
64      image_size: tuple[int, int],
65      file_name: str = "eigenfaces.jpg",
66      num_eigenfaces: int = 25,
67      num_reconstructed_faces: int = 10,
68      cols_per_row: int = 5,
69  ):
70      # D: features
71      # E: num_components
72      # samples: N by D matrix
73      # weights: D by E matrix
74      width, height = image_size
75      os.makedirs(out_dir, exist_ok=True)
76
77      weights = weights[:, :num_eigenfaces]
78      # visualize eigenfaces
79      plot_faces(
80          os.path.join(out_dir, file_name),
81          weights.T.reshape(num_eigenfaces, width, height),
82          cols_per_row,
83      )
84
85      # randomly sample reconstructed_faces
86      indices = np.random.choice(samples.shape[0], num_reconstructed_faces, replace=False)
87      samples = samples[indices]
88
89      # visualize original faces
90      plot_faces(
91          os.path.join(out_dir, f"original_reconstructed_{file_name}"),
92          samples.reshape(num_reconstructed_faces, width, height),
93          cols_per_row,
94      )
95
96      # reconstruct faces
97      faces = samples @ weights @ weights.T
98
99      # visualize reconstructed faces
100     plot_faces(
101         os.path.join(out_dir, f"reconstructed_{file_name}"),
102         faces.reshape(num_reconstructed_faces, width, height),
103         cols_per_row,
104     )
105
106     error = cdist(faces, samples, metric="euclidean").diagonal().mean()
107     print("Reconstruction Average Error: ", error)
108

```

Part2

- `test.evaluate` function: evaluate performance by K-NN algorithm on PCA/LDA features

Arguments

train_data: np.ndarray, shape (N, D), train data samples
train_labels: np.ndarray, shape (N,), train data labels
test_data: np.ndarray, shape (N, D), test data samples
test_labels: np.ndarray, shape (N,), test data labels
pca_weights: PCA weights, shape (D, E)
fisher_weights: LDA weights, shape (D, E)
k_neighbors: int, k-NN

```
135 def evaluate(
136     train_data: npt.NDArray[np.float64],
137     train_labels: npt.NDArray[np.int32],
138     test_data: npt.NDArray[np.float64],
139     test_labels: npt.NDArray[np.int32],
140     pca_weights: npt.NDArray[np.float64],
141     fisher_weights: npt.NDArray[np.float64],
142     k_neighbors: int,
143 ):
144     pca_predictions = classify(
145         train_data, train_labels, test_data, pca_weights, k_neighbors
146     )
147     fisher_predictions = classify(
148         train_data, train_labels, test_data, fisher_weights, k_neighbors
149     )
150
151     print("K-NN Error rate (PCA):", np.mean(pca_predictions != test_labels))
152     print("K-NN Error rate (Fisher):", np.mean(fisher_predictions != test_labels))
```

- `test.classify` function: use K-NN algorithm to classify images

Arguments

train_data: np.ndarray, shape (N, D), train data samples
train_labels: np.ndarray, shape (N,), train data labels
test_data: np.ndarray, shape (N, D), test data samples
weights: PCA or LDA weights, shape (D, E)
k_neighbors: int, k-NN

```

110 ✓ def classify(
111     train_data: npt.NDArray[np.float64],
112     train_labels: npt.NDArray[np.int32],
113     test_data: npt.NDArray[np.float64],
114     weights: npt.NDArray[np.float64],
115     k_neighbors: int = 5,
116 ) → npt.NDArray[np.int32]:
117     # project onto low dimensional space
118     train_data = train_data @ weights
119     test_data = test_data @ weights
120
121     # find labels of k neighbors
122     distance = cdist(test_data, train_data, metric="euclidean")
123     k_indices = np.argsort(distance, axis=-1)[:, :k_neighbors]
124     labels = train_labels[k_indices.flatten()].reshape(-1, k_neighbors)
125
126     # find mode of labels
127     predictions = []
128     for row in labels:
129         classes, counts = np.unique(row, return_counts=True)
130         predictions.append(classes[np.argmax(counts)])
131
132     return np.stack(predictions, axis=0)

```

Part3

- Algorithms

- Kernel PCA

1. Calculate Kernel matrix of data X (N by N matrix), RBF/Linear/Poly

2. Centered Kernel matrix

$$K^C = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$$

3. According to the normal PCA eigenvalue problem in feature space, we could use eigen decomposition on centered Kernel matrix to get the first k largest eigenvectors (principal components) as A matrix

$$Ka = \lambda Na$$

$$\frac{1}{N}Ka = \lambda a$$

4. Project the data onto low dimensional space

$$Z = K(X_{new}, X)A$$

- Kernel LDA

1. Calculate Kernel matrix of data X (N by N matrix), RBF/Linear/Poly

2. Within-class scatter \mathbf{S}_W

$$\mathbf{M}_j = \frac{1}{N_j} \sum_{k \in C_j} k(x_k, x_i), i \in N$$

$$\mathbf{S}_W = \sum_{j=1}^k K_j(I - \mathbf{1}_{n_j})K_j^T$$

3. Between-class scatter \mathbf{S}_B

$$\mathbf{M} = \frac{1}{N} \sum_{k \in N} k(x_k, x_i), i \in N$$

$$\mathbf{S}_B = \sum_{j=1}^k N_j(\mathbf{M}_j - \mathbf{M})(\mathbf{M}_j - \mathbf{M})^T$$

4. Use eigen decomposition on $\mathbf{S}_W^{-1}\mathbf{S}_B$ to get the first k largest eigenvectors (principal components) as A matrix

5. Project the data onto low dimensional space

$$Z = K(X_{new}, X)A$$

- `utils.kernel` function: calculate Kernel matrix

Arguments

x: np.ndarray, shape (N, D), a set of data
y: np.ndarray, shape (N, D), another set of data
kernel_type: Literal["rbf", "linear", "poly"], choose a Kernel to calculate Kernel matrix
center: center it after calculating Kernel matrix

To center the kernel, we could use this formula.

$$K^C = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$$

```
49  def kernel(
50      x: npt.NDArray[np.float64],
51      y: npt.NDArray[np.float64],
52      kernel_type: KERNEL_TYPES,
53      center: bool = False,
54  ) → npt.NDArray[np.float64]:
55      gamma = 1 / x.shape[1]
56      if kernel_type == "rbf":
57          kernel = np.exp(-gamma * cdist(x, y, metric="spherical"))
58      else:
59          kernel = x @ y.T
60          if kernel_type == "poly":
61              kernel = (gamma * kernel) ** 3
62
63      if center:
64          # center the matrix
65          #  $K_C = K - \mathbf{1}_N K - K \mathbf{1}_N + \mathbf{1}_N K \mathbf{1}_N$ 
66          all_inv_n_matrix = np.full_like(kernel, 1 / kernel.shape[0])
67          kernel = (
68              kernel
69              - all_inv_n_matrix @ kernel
70              - kernel @ all_inv_n_matrix
71              + all_inv_n_matrix @ kernel @ all_inv_n_matrix
72          )
73
74  return kernel
```

- `train.py` main

In Part3, we focus on Kernel PCA and LDA.

Kernel PCA assumes the Kernel is centered.

So, before doing eigen decomposition, we have to calculate the Kernel first, then doing dimensional reduction.

Before going to the function, we also need to multiply the inverse of number of data.

$$\frac{1}{N} K$$

```

149 if __name__ == "__main__":
150     args = Arguments().parse_args()
151
152     os.makedirs(args.out_dir, exist_ok=True)
153
154     pattern = os.path.join(args.root_dir, TRAIN_FOLDER, "*.pgm")
155     files = glob.glob(pattern)
156
157     data, labels, _ = load_data(files)
158
159     if args.kernel != "none":
160         # PCA assumes the kernel is centered
161         centered_x = kernel(data, data, args.kernel, center=True) / data.shape[0]
162         x = kernel(data, data, args.kernel)
163
164         pca_weights = solve_by_pca(centered_x, args.kernel, args.num_components)
165         fisher_weights = solve_by_fisher(x, labels, args.kernel, args.num_components)
166         np.save(os.path.join(args.out_dir, "kernel_eigen.npy"), pca_weights)
167         np.save(os.path.join(args.out_dir, "kernel_fisher.npy"), fisher_weights)
168     else:
169         x = np.cov(data, rowvar=False, bias=True)
170
171         pca_weights = solve_by_pca(x, args.kernel, args.num_components)
172         fisher_weights = solve_by_fisher(data, labels, args.kernel, args.num_components)
173         np.save(os.path.join(args.out_dir, "eigen.npy"), pca_weights)
174         np.save(os.path.join(args.out_dir, "fisher.npy"), fisher_weights)
175

```

- `train.solve_by_pca` function: use Kernel PCA algorithm to find A orthogonal projection matrix

Arguments

`x`: np.ndarray, Kernel matrix of data X
`kernel_type`: Literal["rbf", "linear", "poly"], used for checking some requirements
`num_components`: keep first k principal components

In Part3, we only focus on Kernel PCA.

According to the normal PCA eigenvalue problem in feature space, we could use eigen decomposition on centered Kernel matrix to get the first k largest eigenvectors (principal components) as A matrix

$$Ka = \lambda Na$$

$$\frac{1}{N}Ka = \lambda a$$

```

27 def solve_by_pca(
28     x: npt.NDArray[np.float64], kernel_type: KERNEL_TYPES, num_components: int
29 ) -> npt.NDArray[np.float64]:
30     # x: D by D matrix (normal)
31     # x: N by N matrix (kernel)
32
33     if num_components > x.shape[0]:
34         text = "the number of features"
35         if kernel_type != "none":
36             text = "the number of samples"
37         raise ValueError(f"The num_components should not be larger than {text}.")
38
39     start_time = timeit.default_timer()
40
41     eigenvalues, eigenvectors = np.linalg.eigh(x)
42
43     print(
44         "Finished solving eigenvectors in",
45         timeit.default_timer() - start_time,
46         "seconds",
47     )
48
49     # find num_components largest eigenvectors
50     indices = np.argsort(eigenvalues)[-num_components - 1 : -1]
51     eigenvectors = eigenvectors[:, indices]
52     return eigenvectors

```

- `train.calculate_between_and_with_class_covariance` function: calculate between-class and within-class kernel covariance

Arguments

`x`: np.ndarray, Kernel matrix of data X
`labels`: np.ndarray, gt labels of data X
`kernel_type`: Literal["rbf", "linear", "poly"], used for checking some conditions
`num_components`: keep first k principal components

In Part3, we only focus on Kernel LDA.

Here, it corresponds to the second and third step of Kernel LDA.

1. Within-class scatter \mathbf{S}_W

$$\mathbf{M}_j = \frac{1}{N_j} \sum_{k \in C_j} k(x_k, x_i), i \in N$$

$$\mathbf{S}_W = \sum_{j=1}^k K_j(I - \mathbf{1}_{n_j})K_j^T$$

2. Between-class scatter \mathbf{S}_B

$$\mathbf{M} = \frac{1}{N} \sum_{k \in N} k(x_k, x_i), i \in N$$

$$\mathbf{S}_B = \sum_{j=1}^k N_j(\mathbf{M}_j - \mathbf{M})(\mathbf{M}_j - \mathbf{M})^T$$

```

55     def calculate_between_and_with_class_covariance(
56         x: npt.NDArray[np.float64], labels: npt.NDArray[np.int32], kernel_type: KERNEL_TYPES
57     ) -> tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]]:
58         # calculate global mean: normal + kernel
59         global_mean: npt.NDArray[np.float64] = np.mean(x, axis=0, keepdims=True)
60
61         # calculate with-class & between-class covariance
62         groups, counts = np.unique(labels, return_counts=True)
63         covariance_between_class = np.zeros((x.shape[1],) * 2, dtype=np.float64)
64         covariance_within_class = np.zeros_like(covariance_between_class)
65         for i in groups:
66             group_x = x[labels == i]
67             group_mean: npt.NDArray[np.float64] = np.mean(group_x, axis=0, keepdims=True)
68
69             # between-class covariance: normal + kernel
70             centered_group_mean = group_mean - global_mean
71             covariance_between_class += (
72                 counts[i] * centered_group_mean.T @ centered_group_mean
73             )
74
75         if kernel_type == "none":
76             # within-class covariance: normal
77             centered_group_x = group_x - group_mean
78             covariance_within_class += np.dot(centered_group_x.T, centered_group_x)
79         else:
80             # within-class covariance: kernel
81             group_identity = np.identity(counts[i])
82             covariance_within_class += (
83                 group_x.T
84                 @ (group_identity - np.full_like(group_identity, 1 / counts[i]))
85                 @ group_x
86             )
87
88         return covariance_between_class, covariance_within_class

```

- `train.solve_by_fisher` function: use Kernel LDA algorithm to find A orthogonal projection matrix

Arguments

`x`: np.ndarray, Kernel matrix of data X
`labels`: np.ndarray, gt labels of data X
`kernel_type`: Literal["rbf", "linear", "poly"], used for checking some conditions
`num_components`: keep first k principal components

In Part3, we only focus on Kernel LDA.

After calculating the between-class and within-class kernel covariance,

use eigen decomposition on $\mathbf{S}_W^{-1} \mathbf{S}_B$ to get the first k largest eigenvectors (principal components) as A matrix.

```
91     def solve_by_fisher(
92         x: npt.NDArray[np.float64],
93         labels: npt.NDArray[np.int32],
94         kernel_type: KERNEL_TYPES,
95         num_components: int,
96     ) → npt.NDArray[np.float64]:
97         # x: N by D matrix
98         # labels: N array
99
100        if num_components > x.shape[1]:
101            text = "the number of features"
102            if kernel_type ≠ "none":
103                text = "the number of samples"
104            raise ValueError(f"The num_components should not be larger than {text}.")
105
106        start_time = timeit.default_timer()
107        (
108            covariance_between_class,
109            covariance_within_class,
110        ) = calculate_between_and_with_class_covariance(x, labels, kernel_type)
111
112        print(
113            "Finished calculating covariances in",
114            timeit.default_timer() - start_time,
115            "seconds",
116        )
117
118    start_time = timeit.default_timer()
119
120    #  $S_w^{-1} * S_b$ 
121    inversed_covariance_within_class = np.linalg.pinv(covariance_within_class)
122    eigenvalues, eigenvectors = np.linalg.eigh(
123        inversed_covariance_within_class @ covariance_between_class
124    )
125
126    print(
127        "Finished solving eigenvectors in",
128        timeit.default_timer() - start_time,
129        "seconds",
130    )
131
132    # find num_components largest eigenvectors
133    indices = np.argsort(eigenvalues)[:-num_components - 1:-1]
134    eigenvectors = eigenvectors[:, indices]
135    return eigenvectors
```

- `test.py` main: program entry point

```

192 ✓ if __name__ == "__main__":
193     args = Arguments().parse_args()
194
195     # load data
196     train_data, train_labels, image_size = load_dataset(args.root_dir, TRAIN_FOLDER)
197     test_data, test_labels, _ = load_dataset(args.root_dir, TEST_FOLDER)
198
199     # load weights
200     pca_weights = np.load(args.eigen_path)
201     fisher_weights = np.load(args.fisher_path)
202
203     if args.kernel == "none":
204         visualize_faces(
205             args.out_dir,
206             train_data,
207             pca_weights,
208             image_size,
209             num_eigenfaces=args.num_eigenfaces,
210             num_reconstructed_faces=args.num_reconstructed_faces,
211             file_name="eigenfaces.jpg",
212         )
213     else:
214         visualize_faces(
215             args.out_dir,
216             train_data,
217             fisher_weights,
218             image_size,
219             num_eigenfaces=args.num_eigenfaces,
220             num_reconstructed_faces=args.num_reconstructed_faces,
221             file_name="fisherfaces.jpg",
222         )
223     evaluate(
224         train_data,
225         train_labels,
226         test_data,
227         test_labels,
228         pca_weights,
229         fisher_weights,
230         args.k_neighbors,
231     )
232 else:
233     evaluate_kernel(
234         train_data,
235         train_labels,
236         test_data,
237         test_labels,
238         pca_weights,
239         fisher_weights,
240         args.k_neighbors,
241     )
242

```

- `test.evaluate_kernel` function: evaluate performance by K-NN algorithm on Kernel PCA/LDA features

Arguments

train_data: np.ndarray, shape (N, D), train data samples
 train_labels: np.ndarray, shape (N,), train data labels
 test_data: np.ndarray, shape (N, D), test data samples
 test_labels: np.ndarray, shape (N,), test data labels
 pca_weights: Kernel PCA weights, shape (D, E)
 fisher_weights: Kernel LDA weights, shape (D, E)
 k_neighbors: int, k-NN

Before classifying the test data, we have to calculate kernel matrix.

```

55     def evaluate_kernel(
56         train_data: npt.NDArray[np.float64],
57         train_labels: npt.NDArray[np.int32],
58         test_data: npt.NDArray[np.float64],
59         test_labels: npt.NDArray[np.int32],
60         pca_weights: npt.NDArray[np.float64],
61         fisher_weights: npt.NDArray[np.float64],
62         k_neighbors: int,
63         kernel_type: KERNEL_TYPES,
64     ):
65         # calculate kernels
66         train_kernel = kernel(train_data, train_data, kernel_type)
67         test_kernel = kernel(test_data, train_data, kernel_type)
68
69         pca_predictions = classify(
70             train_kernel,
71             train_labels,
72             test_kernel,
73             pca_weights,
74             k_neighbors,
75         )
76         fisher_predictions = classify(
77             train_kernel, train_labels, test_kernel, fisher_weights, k_neighbors
78         )

```

- `test.classify` function: use K-NN algorithm to classify images

Arguments

`train_data`: np.ndarray, shape (N, N), kernel matrix of train data
`train_labels`: np.ndarray, shape (N,), train data labels
`test_data`: np.ndarray, shape (K, N), kernel matrix of test data
`weights`: Kernel PCA or LDA weights, shape (N, E)
`k_neighbors`: int, k-NN

```

110     def classify(
111         train_data: npt.NDArray[np.float64],
112         train_labels: npt.NDArray[np.int32],
113         test_data: npt.NDArray[np.float64],
114         weights: npt.NDArray[np.float64],
115         k_neighbors: int = 5,
116     ) → npt.NDArray[np.int32]:
117         # project onto low dimensional space
118         train_data = train_data @ weights
119         test_data = test_data @ weights
120
121         # find labels of k neighbors
122         distance = cdist(test_data, train_data, metric="euclidean")
123         k_indices = np.argsort(distance, axis=-1)[:, :k_neighbors]
124         labels = train_labels[k_indices.flatten()].reshape(-1, k_neighbors)
125
126         # find mode of labels
127         predictions = []
128         for row in labels:
129             classes, counts = np.unique(row, return_counts=True)
130             predictions.append(classes[np.argmax(counts)])
131
132     return np.stack(predictions, axis=0)

```

t-SNE

Part1

- Algorithms

- Symmetric SNE

1. In practice, we calculate the conditional probability $p_{j|i}$ first in the original dimensional space.

We also need to choose N precisions for conditional probabilities to be the same perplexity (entropy) by using binary search.

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

$$Perp(P_i) = 2^{H(P_i)} \quad \text{perplexity}$$

$$H(P_i) = - \sum_j p_{j|i} \log_2 p_{j|i} \quad \text{entropy}$$

2. Calculate the joint probability p_{ij}

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

3. Calculate the joint probability q_{ij} in the low dimensional space (gaussian distribution)

At the initial time, we need to randomly initialize y

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

4. Calculate the gradient for each y_i

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

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j).$$

5. Use optimization algorithm to update y_i

6. Repeat 3rd, 4th and 5th steps until achieving the maximum loop or coverage

- t-SNE

1. In practice, we calculate the conditional probability $p_{j|i}$ first in the original dimensional space.

We also need to choose N precisions for conditional probabilities to be the same perplexity (entropy) by using binary search.

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

$$Perp(P_i) = 2^{H(P_i)} \quad \text{perplexity}$$

$$H(P_i) = - \sum_j p_{j|i} \log_2 p_{j|i} \quad \text{entropy}$$

2. Calculate the joint probability p_{ij}

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

3. Calculate the joint probability q_{ij} in the low dimensional space (student t-distribution)

At the initial time, we need to randomly initialize y

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

4. Calculate the gradient for each y_i

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

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

5. Use optimization algorithm to update y_i

6. Repeat 3rd, 4th and 5th steps until achieving the maximum loop or coverage

- `sne` function: initialize variables, such as variables for optimization algorithm, variable Y for low dimensional space...

About PCA algorithm, the authors mention it in the paper, [Visualizing Data using t-SNE](#).

So, they want to speed up the computation of pairwise distances between the datapoints.

In all of our experiments, we start by using `PCA` to reduce the dimensionality of the data to 30. This speeds up the computation of pairwise distances between the datapoints and suppresses some noise without severely distorting the interpoint distances. We then use each of the dimensionality

```

177     def sne(
178         X=np.array([]),
179         no_dims=2,
180         initial_dims=50,
181         perplexity=30.0,
182         mode: Mode = Mode.symmetric_sne,
183     ):
184     """
185     Runs t-SNE on the dataset in the NxD array X to reduce its
186     dimensionality to no_dims dimensions. The syntax of the function is
187     `Y = tsne.tsne(X, no_dims, perplexity)`, where X is an NxD NumPy array.
188     """
189
190     # Check inputs
191     if not np.issubdtype(X.dtype, np.floating):
192         print("Error: array X should be a float array.")
193         return -1
194     if isinstance(no_dims, float):
195         print("Error: number of dimensions should be an integer.")
196         return -1
197
198     # Initialize variables
199     X = pca(X, initial_dims).real
200     (n, d) = X.shape
201     max_iter = 1000
202     initial_momentum = 0.5
203     final_momentum = 0.8
204     eta = 500
205     min_gain = 0.01
206     Y = np.random.randn(n, no_dims)
207     dY = np.zeros((n, no_dims))
208     iY = np.zeros((n, no_dims))
209     gains = np.ones((n, no_dims))
210     history_images: list[Image.Image] = []

```

- `x2p` function: calculate $p_{i|j}$ conditional probability

Here, it corresponds to the first step of s-SNE and t-SNE.

In practice, we calculate the conditional probability $p_{j|i}$ first in the original dimensional space.

We also need to choose N precisions for conditional probabilities to be the same perplexity (entropy) by using **binary search**.

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

$$Perp(P_i) = 2^{H(P_i)} \quad \text{perplexity}$$

$$H(P_i) = - \sum_j p_{j|i} \log_2 p_{j|i} \quad \text{entropy}$$

```

105 ~ def k2p(X=np.array([]), tol=1e-5, perplexity=30.0):
106 ~ """
107 ~     Performs a binary search to get P-values in such a way that each
108 ~     conditional Gaussian has the same perplexity.
109 ~ """
110
111     # Initialize some variables
112     print("Computing pairwise distances ... ")
113     (n, d) = X.shape
114     sum_X = np.sum(np.square(X), 1)
115     D = np.add(np.add(-2 * np.dot(X, X.T), sum_X).T, sum_X)
116     P = np.zeros((n, n))
117     beta = np.ones((n, 1))
118     logU = np.log(perplexity)
119
120     # Loop over all datapoints
121     for i in range(n):
122         # Print progress
123         if i % 500 == 0:
124             print("Computing P-values for point %d of %d..." % (i, n))
125
126         # Compute the Gaussian kernel and entropy for the current precision
127         betamin = -np.inf
128         betamax = np.inf
129         Di = D[i, np.concatenate((np.r_[0:i], np.r_[i + 1 : n]))]
130         (H, thisP) = Hbeta(Di, beta[i])
131
132         # Evaluate whether the perplexity is within tolerance
133         Hdiff = H - logU
134         tries = 0
135         while np.abs(Hdiff) > tol and tries < 50:
136             # If not, increase or decrease precision
137             if Hdiff > 0:
138                 betamin = beta[i].copy()
139                 if betamax == np.inf or betamax == -np.inf:
140                     beta[i] = beta[i] * 2.0
141                 else:
142                     beta[i] = (beta[i] + betamax) / 2.0
143             else:
144                 betamax = beta[i].copy()
145                 if betamin == np.inf or betamin == -np.inf:
146                     beta[i] = beta[i] / 2.0
147                 else:
148                     beta[i] = (beta[i] + betamin) / 2.0
149
150             # Recompute the values
151             (H, thisP) = Hbeta(Di, beta[i])
152             Hdiff = H - logU
153             tries += 1
154
155         # Set the final row of P
156         P[i, np.concatenate((np.r_[0:i], np.r_[i + 1 : n]))] = thisP
157
158     # Return final P-matrix
159     print("Mean value of sigma: %f" % np.mean(np.sqrt(1 / beta)))
160     return P

```

- `sne` function: after calculating the conditional probability p_{ij} , then calculate joint probability p_{ij}

Here, it corresponds to the second step of s-SNE and t-SNE.

Calculate the joint probability p_{ij}

$$p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$$

Note, you may notice that the denominator part is `np.sum(P)`.

Because we already calculated the element-wise sum between $p_{j|i}$ and $p_{i|j}$ and also sum each row of $p_{j|i}$ or $p_{i|j}$ is equal to 1, the authors just sum them all as $2N$.

(In my opinion, it is a little waste of computation. We could just calculate it with the shape size. I don't know why they do it like this.)

The early exaggeration method is just used for optimization. So, I don't explain it detailed. You could check their paper.

The last line is used for numerical stability to avoid zero value.

```

212      # Compute P-values
213      P = x2p(X, 1e-5, perplexity)
214      P = P + np.transpose(P)
215      P = P / np.sum(P)
216      P = P * 4.0  # early exaggeration
217      P = np.maximum(P, 1e-12)
218

```

- `sne` function: optimize low dimensional feature y_i

Here, it corresponds to the 3rd, 4th and 5th step of s-SNE and t-SNE.

3rd step (Compute pairwise affinities)

- s-SNE (gaussian distribution)

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

- t-SNE (student t-distribution)

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

4th step (Compute gradient)

- s-SNE

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j).$$

- t-SNE

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

5th step

Use optimization algorithm to update y_i

```

219      # Run iterations
220  ↘  for iter in range(max_iter):
221      | # Compute pairwise affinities
222      | sum_Y = np.sum(np.square(Y), 1)
223      | num = -2.0 * np.dot(Y, Y.T)
224  ↘  | if mode == Mode.symmetric_sne:
225  | | num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
226  ↘  | else:
227  | | num = 1.0 / (1.0 + np.add(np.add(num, sum_Y).T, sum_Y))
228  | | num[range(n), range(n)] = 0.0
229  | | Q = num / np.sum(num)
230  | | Q = np.maximum(Q, 1e-12)
231
232      # Compute gradient
233      PQ = P - Q
234  ↘  for i in range(n):
235  ↘  | if mode == Mode.symmetric_sne:
236  | | dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
237  ↘  | else:
238  | | dY[i, :] = np.sum(
239  | |     np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0
240  | | )
241
242      # Perform the update
243  ↘  if iter < 20:
244  | | momentum = initial_momentum
245  ↘  else:
246  | | momentum = final_momentum
247  | gains = (gains + 0.2) * ((dY > 0.0) != (iY > 0.0)) + (gains * 0.8) * (
248  |     (dY > 0.0) == (iY > 0.0)
249  | )
250  | gains[gains < min_gain] = min_gain
251  | iY = momentum * iY - eta * (gains * dY)
252  | Y = Y + iY
253  | Y = Y - np.tile(np.mean(Y, 0), (n, 1))
254
255      # Compute current value of cost function
256  ↘  if (iter + 1) % 10 == 0:
257  | | history_images.append(draw_results(iter + 1, Y, labels, perplexity, mode))
258  | | C = np.sum(P * np.log(P / Q))
259  | | print("Iteration %d: error is %f" % (iter + 1, C))
260
261      # Stop lying about P-values
262  ↘  if iter == 100:
263  | | P = P / 4.0
264

```

Part2

- `sne` function: record optimization procedure every 10 iterations

```

# Compute current value of cost function
if (iter + 1) % 10 == 0:
    history_images.append(draw_results(iter + 1, Y, labels, perplexity, mode))
    C = np.sum(P * np.log(P / Q))
    print("Iteration %d: error is %f" % (iter + 1, C))

```

- `draw_results` function: draw low dimensional points Y on the figure

```

72 |     def draw_results(
73 |         step: int, Y: np.ndarray, labels: np.ndarray, perplexity: float, mode: Mode
74 |     ):
75 |         fig = pylab.figure()
76 |         ax1 = fig.subplots(1, 1)
77 |
78 |         fig.suptitle(f"Step = {step}, Algorithm = {str(mode)}, Perplexity = {perplexity}")
79 |
80 |         ax1.scatter(Y[:, 0], Y[:, 1], 20, labels)
81 |
82 |         fig.canvas.draw()
83 |         image = Image.frombytes(
84 |             "RGB", fig.canvas.get_width_height(), fig.canvas.tostring_rgb()
85 |         )
86 |
87 |         pylab.close(fig)
88 |         return image
89

```

- `sne` function: save the results in gif format

```

261 |     # Stop lying about P-values
262 |     if iter == 100:
263 |         |     P = P / 4.0
264 |
265 |     history_images[0].save(
266 |         os.path.join(OUT_DIR, f"{str(mode)}_history.gif"),
267 |         save_all=True,
268 |         append_images=history_images[1:],
269 |         optimize=False,
270 |         loop=0,
271 |         duration=300,
272 |     )
273 |     visualize_similarities(P, Q, labels, perplexity, mode)
274 |
275 |     # Return solution
276 |     return Y

```

Part3

- `sne` function: visualize similarities p_{ij} and q_{ij}

```

261 |     # Stop lying about P-values
262 |     if iter == 100:
263 |         |     P = P / 4.0
264 |
265 |     history_images[0].save(
266 |         os.path.join(OUT_DIR, f"{str(mode)}_history.gif"),
267 |         save_all=True,
268 |         append_images=history_images[1:],
269 |         optimize=False,
270 |         loop=0,
271 |         duration=300,
272 |     )
273 |     visualize_similarities(P, Q, labels, perplexity, mode)
274 |
275 |     # Return solution
276 |     return Y

```

- `visualize_similarities` function: visualize similarities p_{ij} and q_{ij}

First, we sort the similarity matrix with gt labels in order to watch the relationship between the cluster and the gt labels.

Then, normalize the similarity matrix by min-max normalization algorithm to keep their original relative similarity.

```
34 |     def visualize_similarities(
35 |         P: np.ndarray, Q: np.ndarray, labels: np.ndarray, perplexity: float, mode: Mode
36 |     ):
37 |         # reorder rows and cols by labels
38 |         # block-wise similarity matrix
39 |         indices = np.argsort(labels)
40 |         P = P[indices][:, indices]
41 |         Q = Q[indices][:, indices]
42 |
43 |         min_P = np.min(P)
44 |         scaled_P = (P - min_P) / (np.max(P) - min_P)
45 |
46 |         min_Q = np.min(Q)
47 |         scaled_Q = (Q - min_Q) / (np.max(Q) - min_Q)
48 |
49 |         fig = pylab.figure()
50 |         ax1, ax2 = fig.subplots(1, 2)
51 |
52 |         fig.suptitle(f"Algorithm = {str(mode)}, Perplexity = {perplexity}")
53 |
54 |         ax1.set_title("High-dimensional space")
55 |         image = ax1.matshow(scaled_P, cmap="hot")
56 |         ax1.set_xlabel("label")
57 |         ax1.set_ylabel("label")
58 |         fig.colorbar(image)
59 |
60 |         ax2.set_title("Low-dimensional space")
61 |         image = ax2.matshow(scaled_Q, cmap="hot")
62 |         ax2.set_xlabel("label")
63 |         ax2.set_ylabel("label")
64 |         fig.colorbar(image)
65 |
66 |         fig.tight_layout()
67 |         fig.savefig(os.path.join(OUT_DIR, f"{str(mode)}_similarities.png"))
68 |
69 |         pylab.close(fig)
```

Part4

- `tsne.py` script: entry point

Use Enum class to control which mode I would like to use.

Use arguments to control the algorithm and perplexity value.

```
26 |     class Mode(Enum):
27 |         symmetric_sne = "s-sne"
28 |         t_sne = "t-sne"
29 |
30 |         def __str__(self):
31 |             return self.value
32 |
```

```

279 if __name__ == "__main__":
280     parser = argparse.ArgumentParser(
281         formatter_class=argparse.ArgumentDefaultsHelpFormatter
282     )
283     parser.add_argument(
284         "--algorithm",
285         type=Mode,
286         choices=list(Mode),
287         default=Mode.symmetric_sne,
288         help="Choose an algorithm to perform dimensional reduction",
289     )
290     parser.add_argument(
291         "--perplexity",
292         type=float,
293         default=20.0,
294         help="perplexity",
295     )
296     args = parser.parse_args()
297
298     print("Running example on 2,500 MNIST digits ... ")
299     X = np.loadtxt("data/mnist2500_X.txt")
300     labels = np.loadtxt("data/mnist2500_labels.txt")
301
302     os.makedirs(OUT_DIR, exist_ok=True)
303
304     Y = sne(
305         X, no_dims=2, initial_dims=50, perplexity=args.perplexity, mode=args.algorithm
306     )
307     pylab.title(str(args.algorithm))
308     pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
309     pylab.savefig(os.path.join(OUT_DIR, f"{str(args.algorithm)}_final.png"))
310     pylab.close()
311

```

Experiments settings and results & Discussion

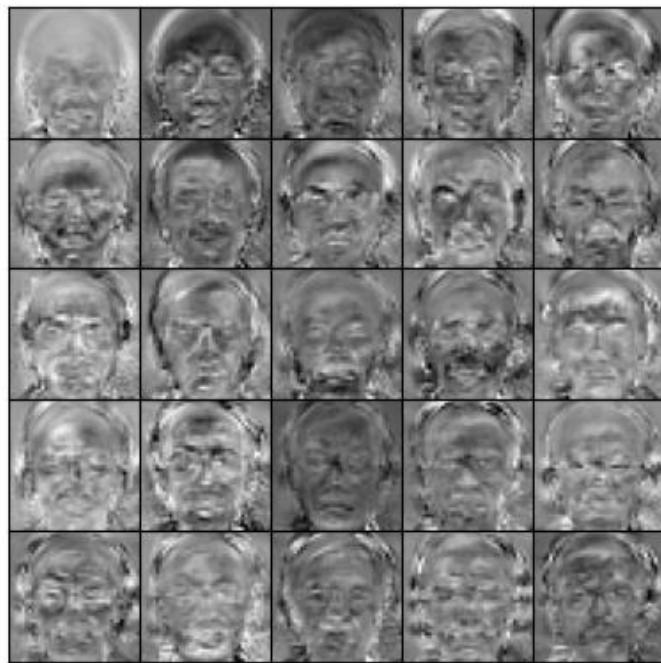
Kernel Eigenfaces

Part1

- Eigenfaces



- Fisherfaces



- Reconstructed faces from eigenfaces





- Reconstructed faces from fisherfaces





Part2

K-NN algorithm: PCA, LDA (Fisher)

```
K-NN Error rate (PCA): 0.1
K-NN Error rate (Fisher): 0.06666666666666667
```

Part3

Observations

- Without kernel, it is better than others with kernel.
- LDA algorithm performs best.
- It is possible that the kernel function need more finetuning, e.g. grid search, to find the best parameters.
- As you can see the results with linear and polynomial kernel, they are very similar.
It makes sense, because the linear kernel is a part of polynomial kernel.
- The results with RBF kernel is better than other two kernels'.
It means that the data in feature space is very likely a gaussian distribution.

Results

- RBF Kernel

```
K-NN Error rate (PCA): 0.16666666666666666
K-NN Error rate (Fisher): 0.2
```

- Linear Kernel

```
K-NN Error rate (PCA): 0.2333333333333334  
K-NN Error rate (Fisher): 0.3
```

- Polynomial Kernel

```
K-NN Error rate (PCA): 0.2333333333333334  
K-NN Error rate (Fisher): 0.2666666666666666
```

t-SNE

Part1

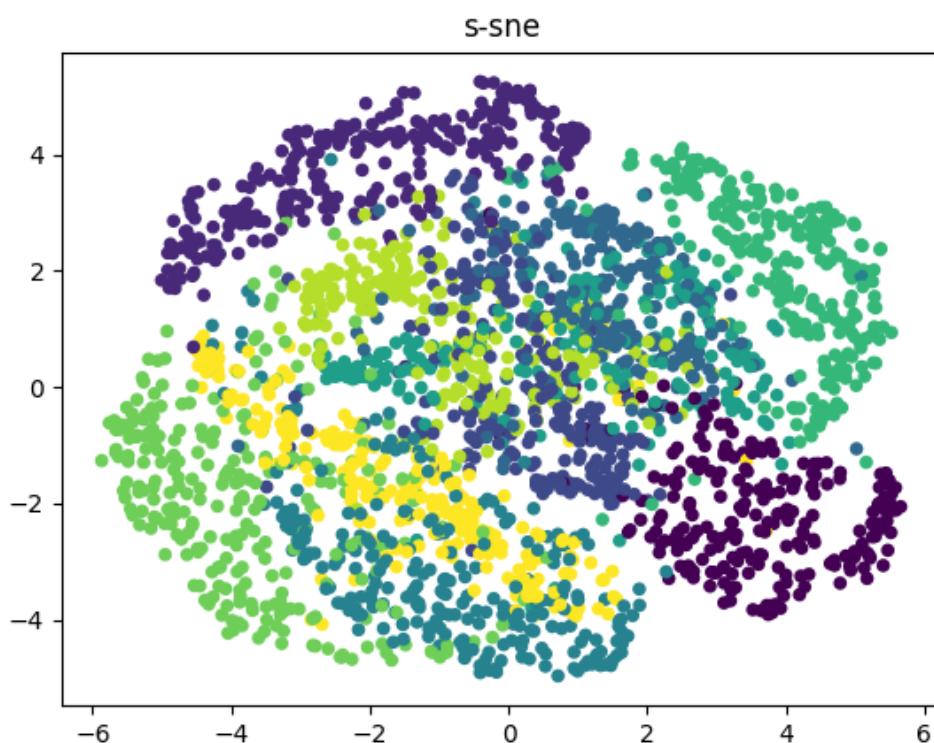
Observations

From the results of s-SNE and t-SNE, we can know that the s-SNE has crowded problem (data points with different labels are highly overlapped).

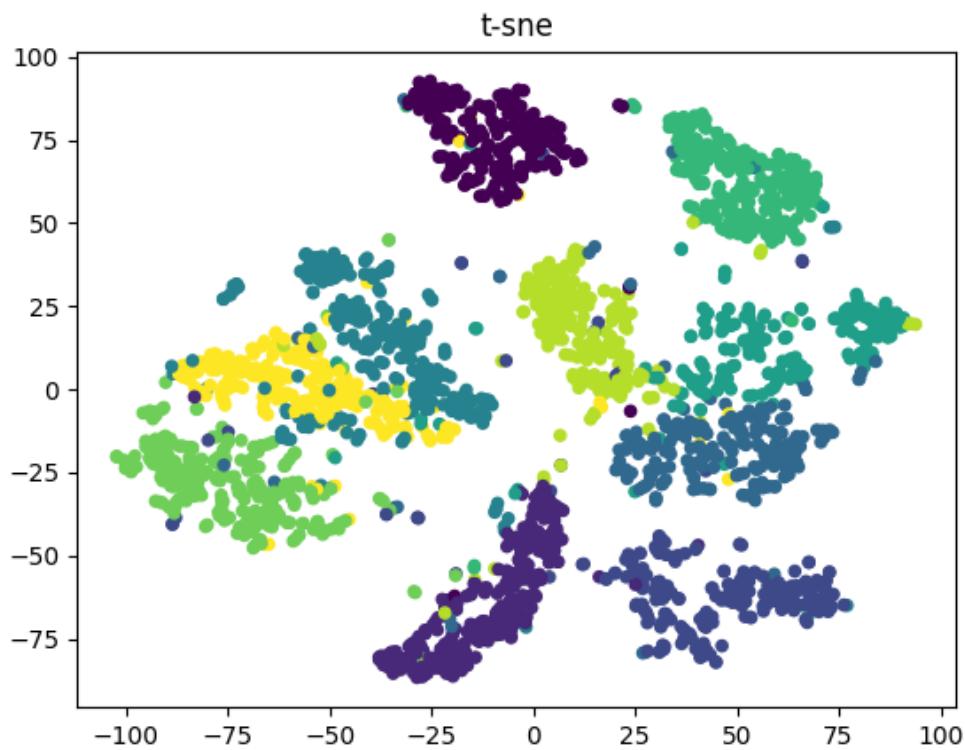
So, in the low dimensional space, it proves that t-SNE uses student t-distribution to formulate the data points better than s-SNE's.

Results

- Symmetric SNE



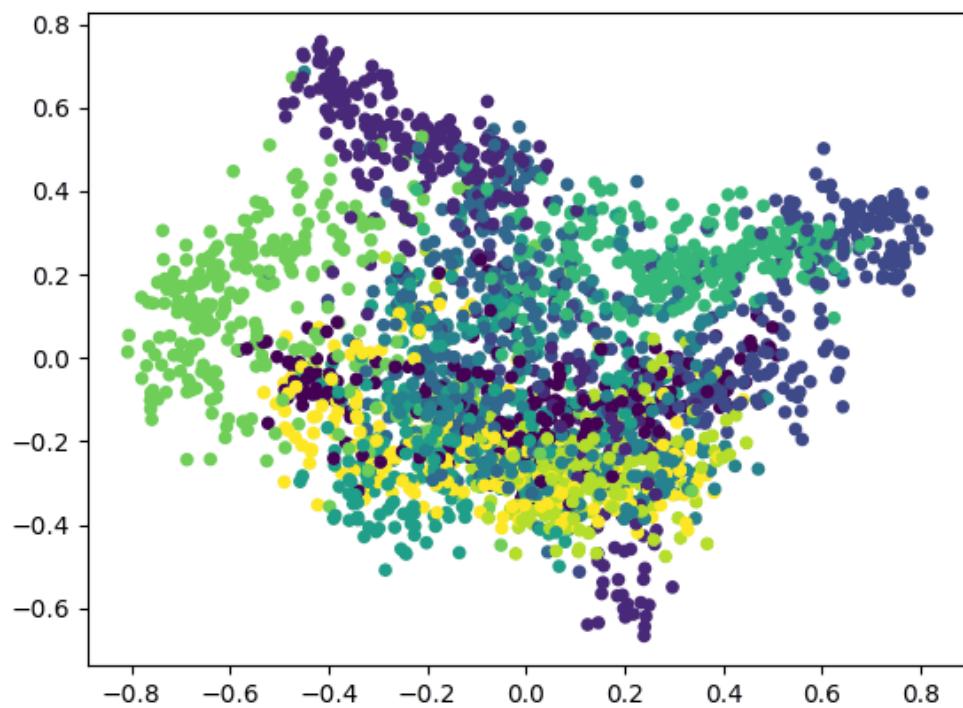
- t-SNE



Part2

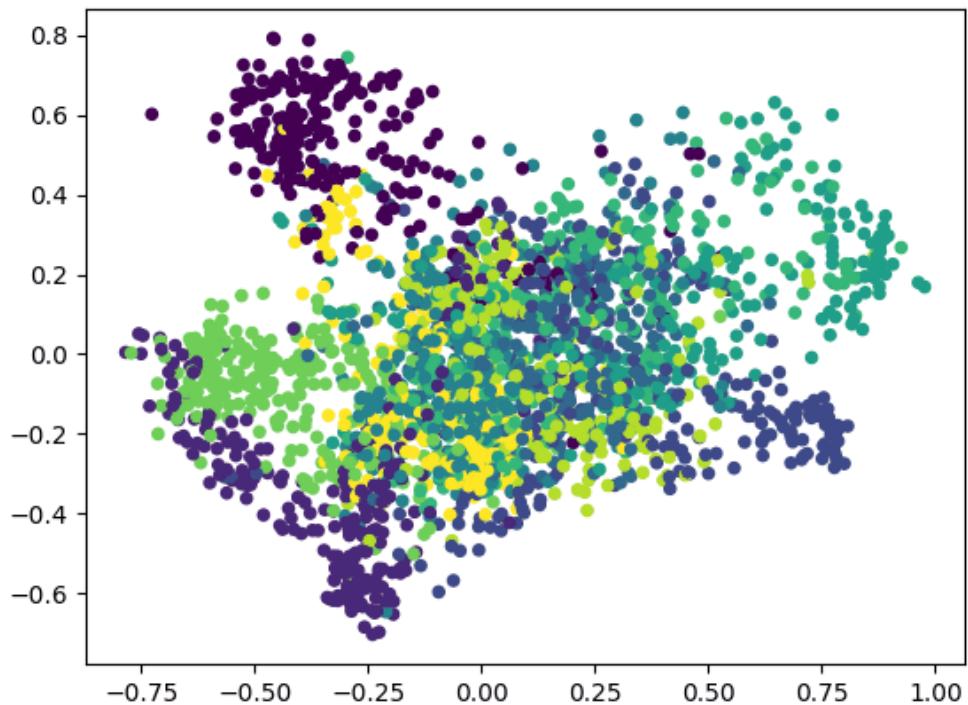
- s-SNE ([imgur gif](#))

Step = 10, Algorithm = s-sne, Perplexity = 20.0



- t-SNE ([imgur gif](#))

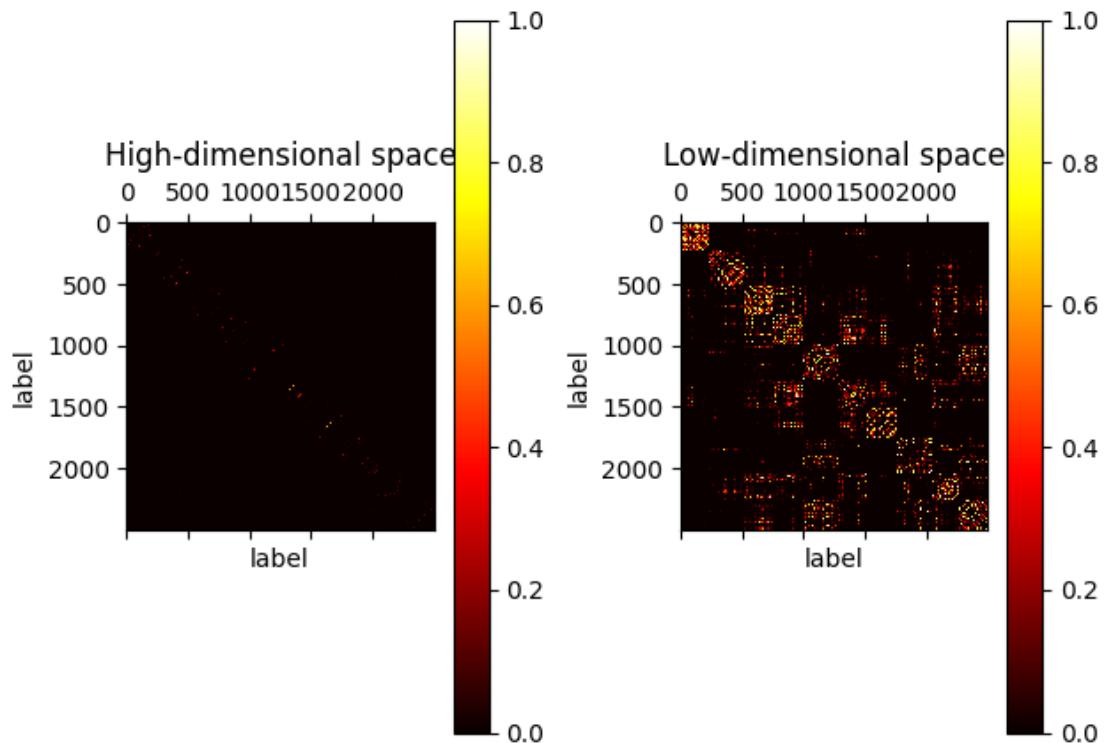
Step = 10, Algorithm = t-sne, Perplexity = 20.0



Part3

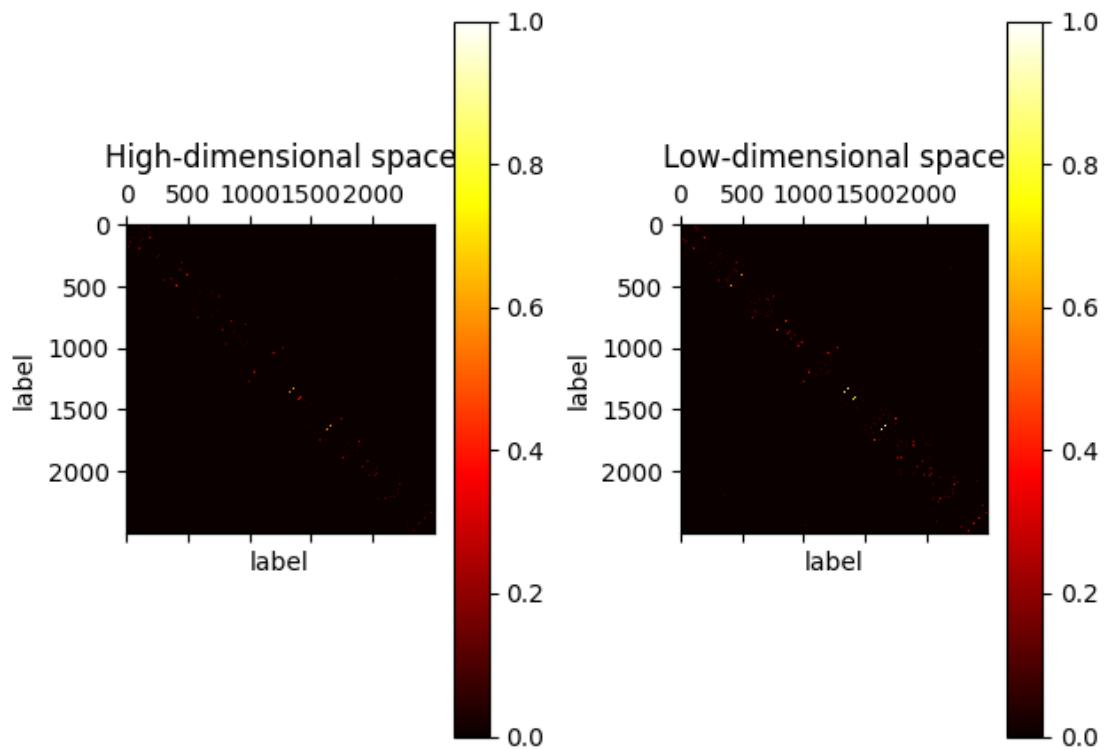
- s-SNE (too crowded in low dimensional space!)

Algorithm = s-sne, Perplexity = 20.0



- t-SNE

Algorithm = t-sne, Perplexity = 20.0



Part4

Observations

- In these points figures, we could know
 - when the perplexity is low (entropy is low), the variance of each group is very small.
 - when the perplexity is high (entropy is high), the variance of each group is very wide.
 - It proves that the perplexity will affect the choose of variance.
- The t-SNE figures always have a good ability to separate groups no matter what the perplexity is.
- From these s-SNE figures, it is hard to see any difference between them.

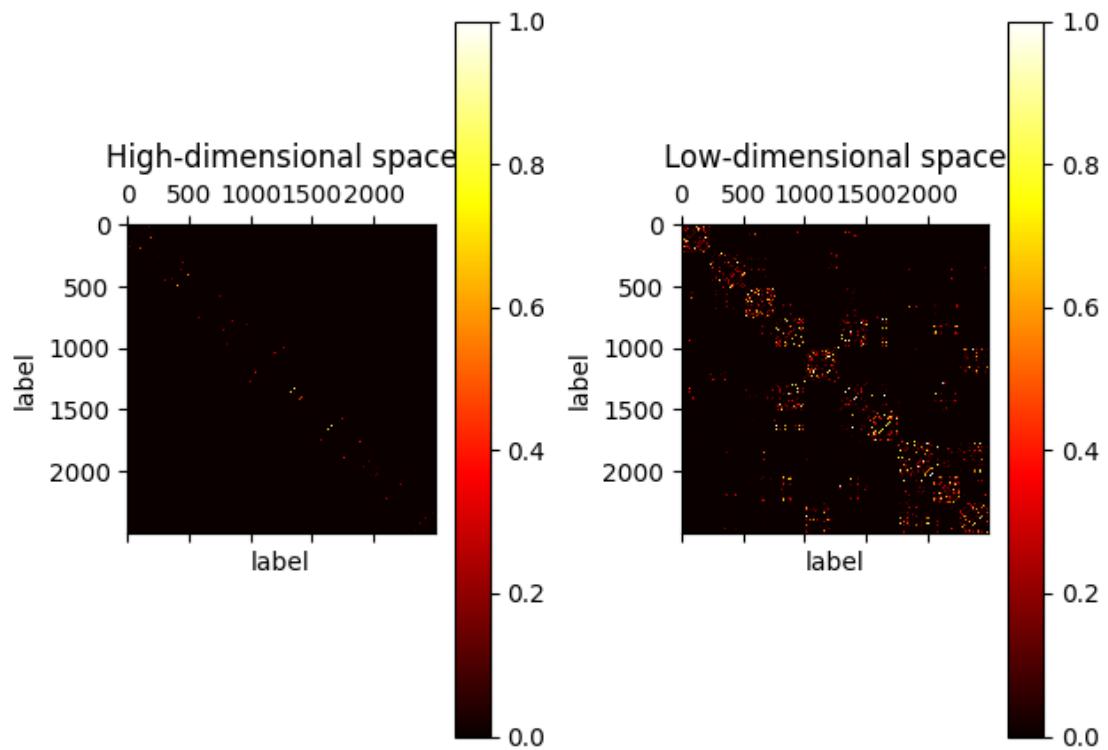
But if we look from the similarity figures, we could see the difference.

For instance, the perplexity is 5 and 35.

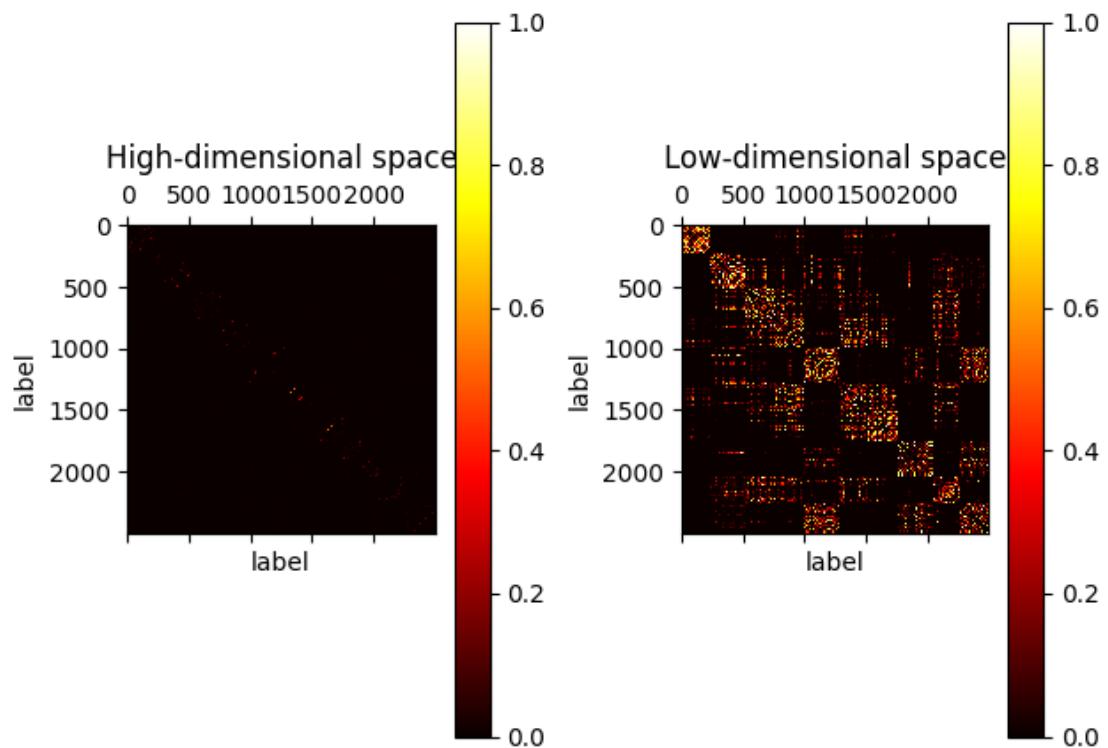
Look at the low dimensional space, because of the crowded problem, we could see the data points are very close.

When the perplexity is going higher, the variance is wider. It is very obvious that the crowded problem is going worse.

Algorithm = s-sne, Perplexity = 5.0



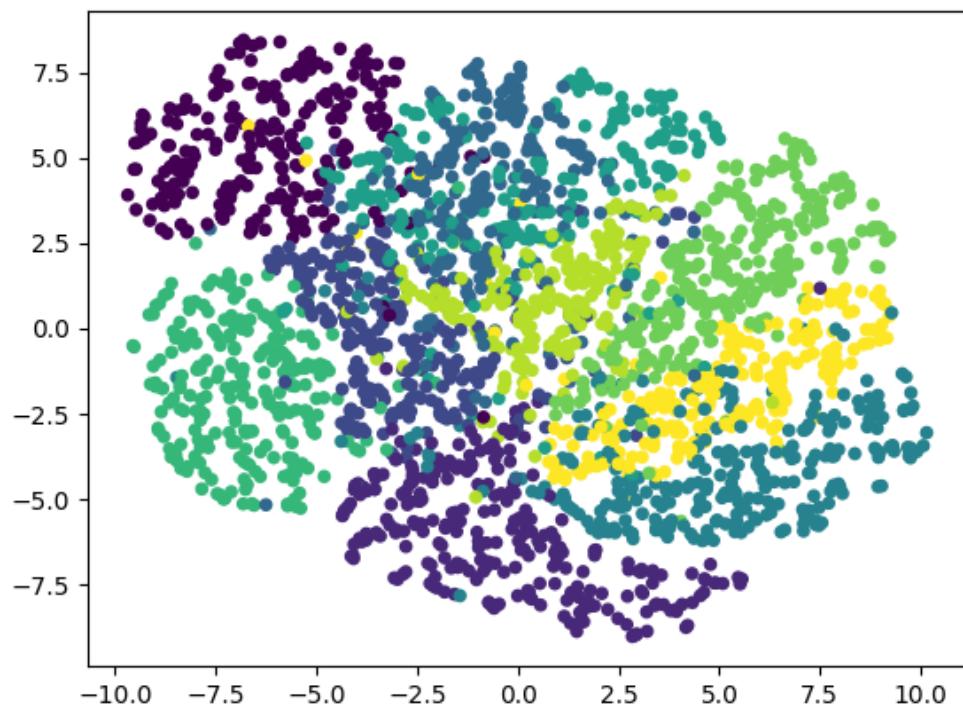
Algorithm = s-sne, Perplexity = 35.0



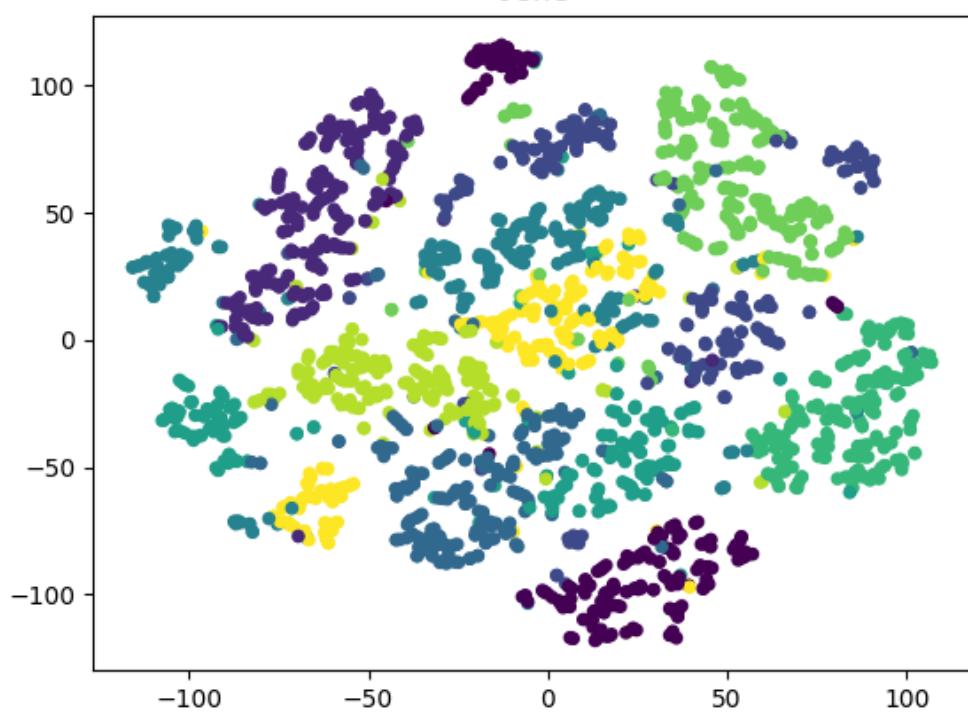
Results

- Perplexity = 5 ([imgur](#), [imgur](#))

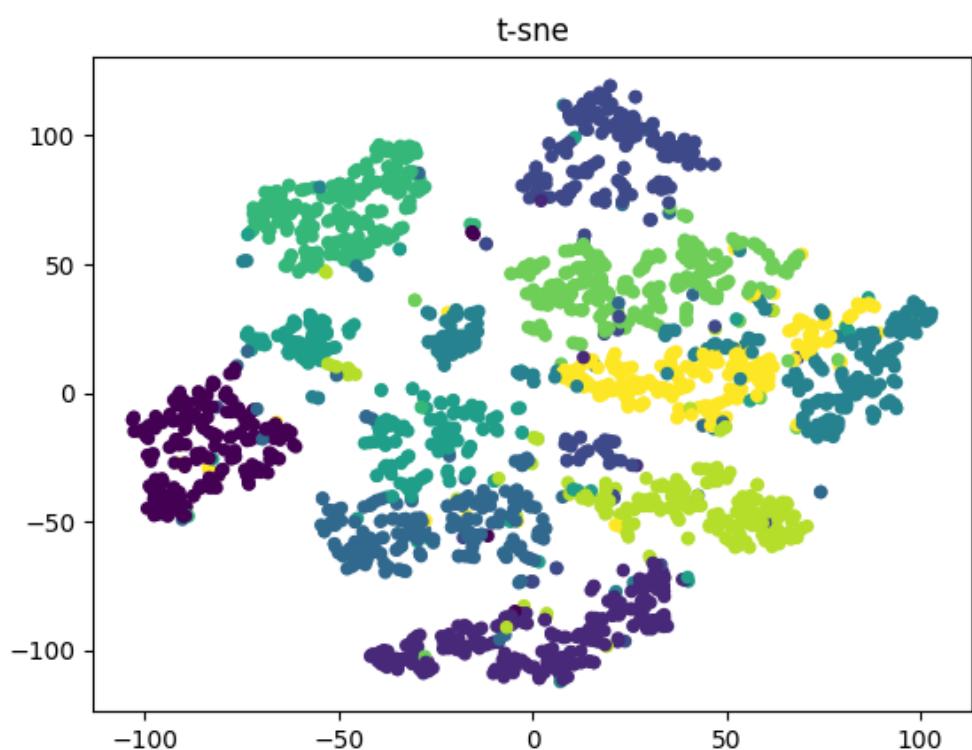
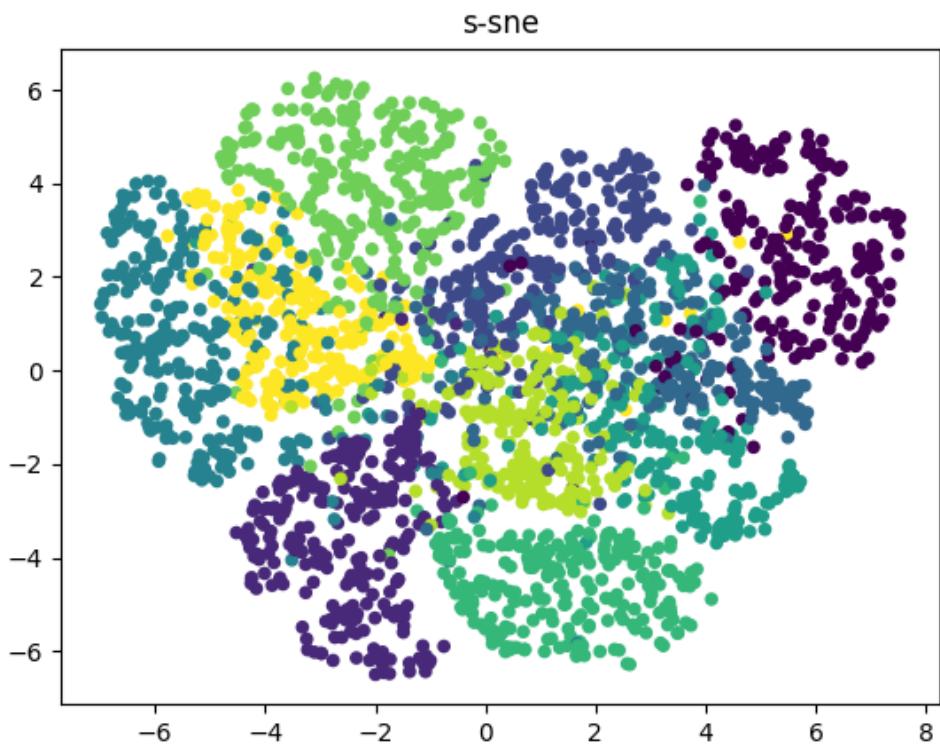
s-sne



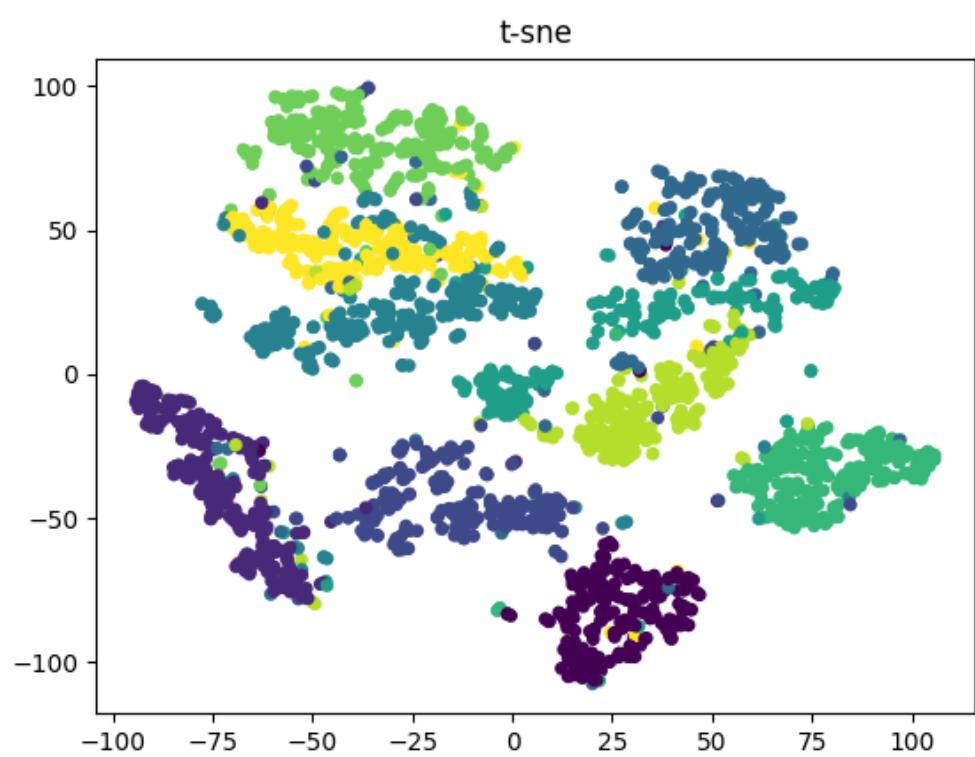
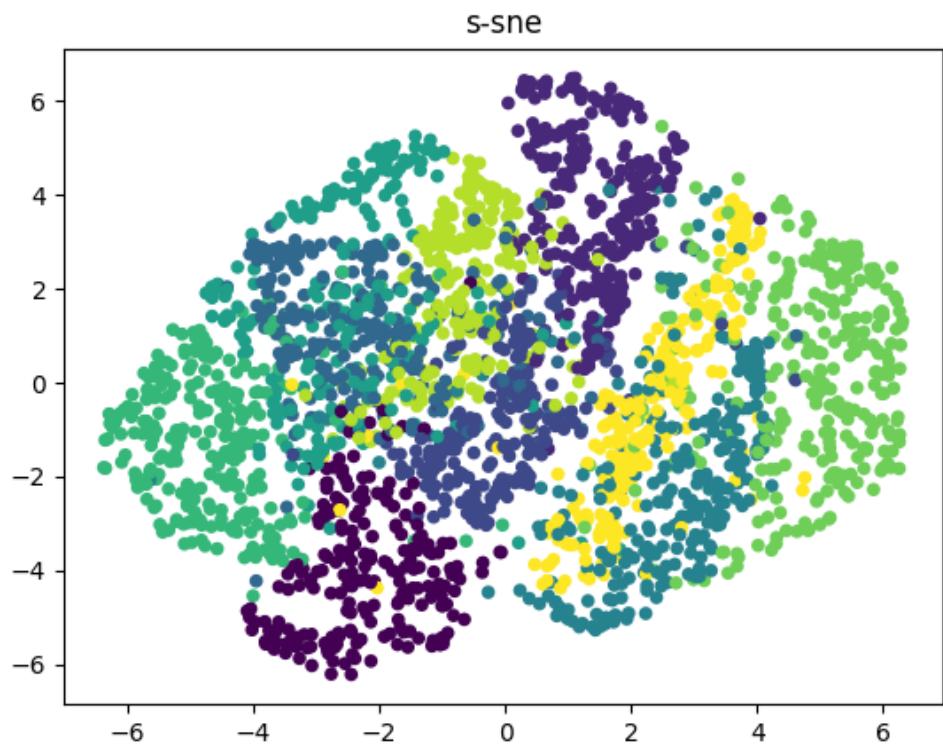
t-sne



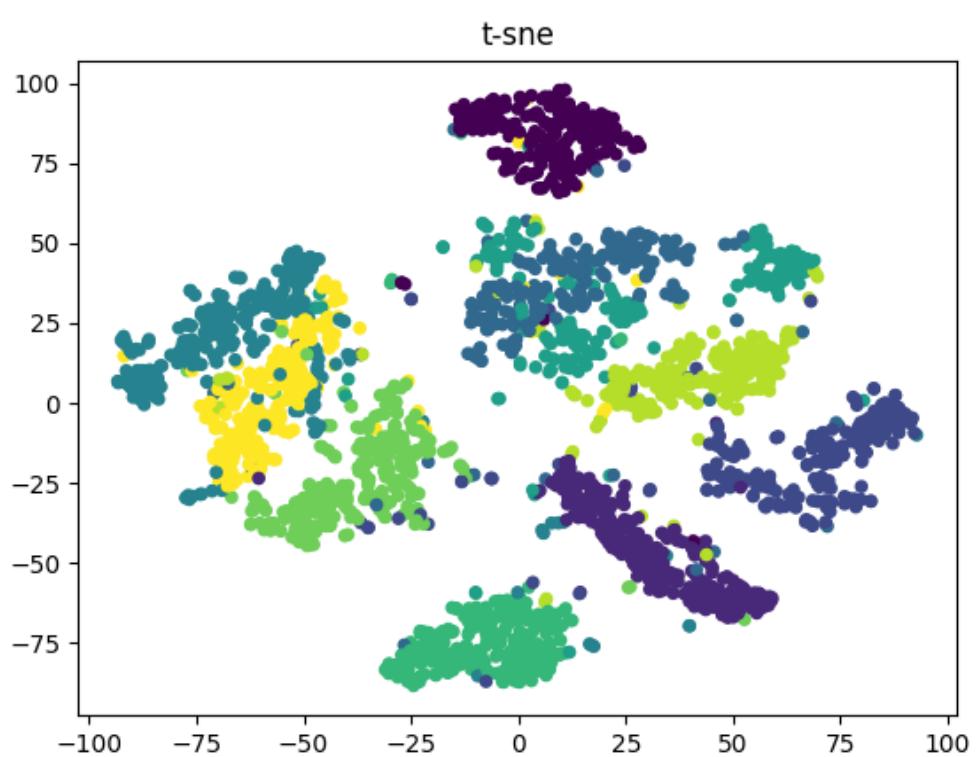
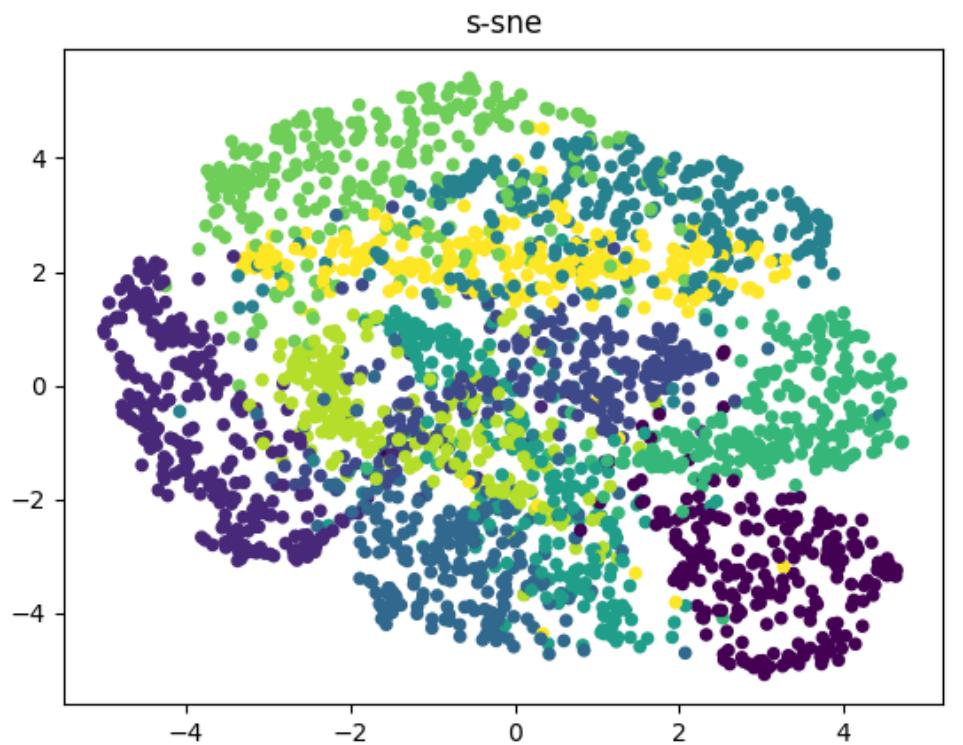
- Perplexity = 10 ([imgur](#), [imgur](#))



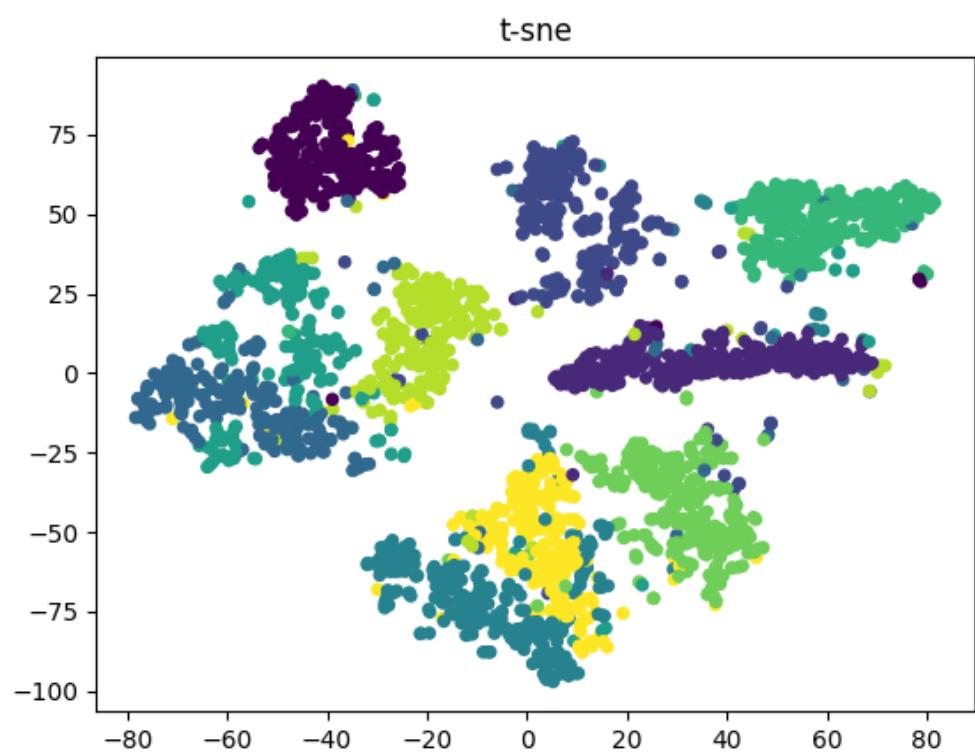
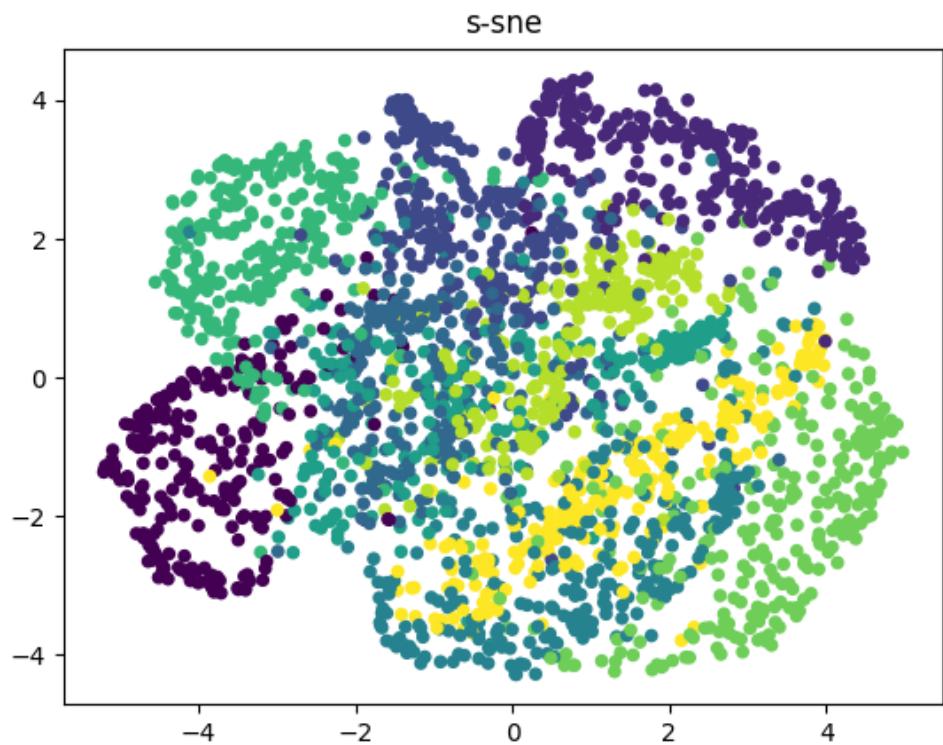
- Perplexity = 15 ([imgur](#), [imgur](#))



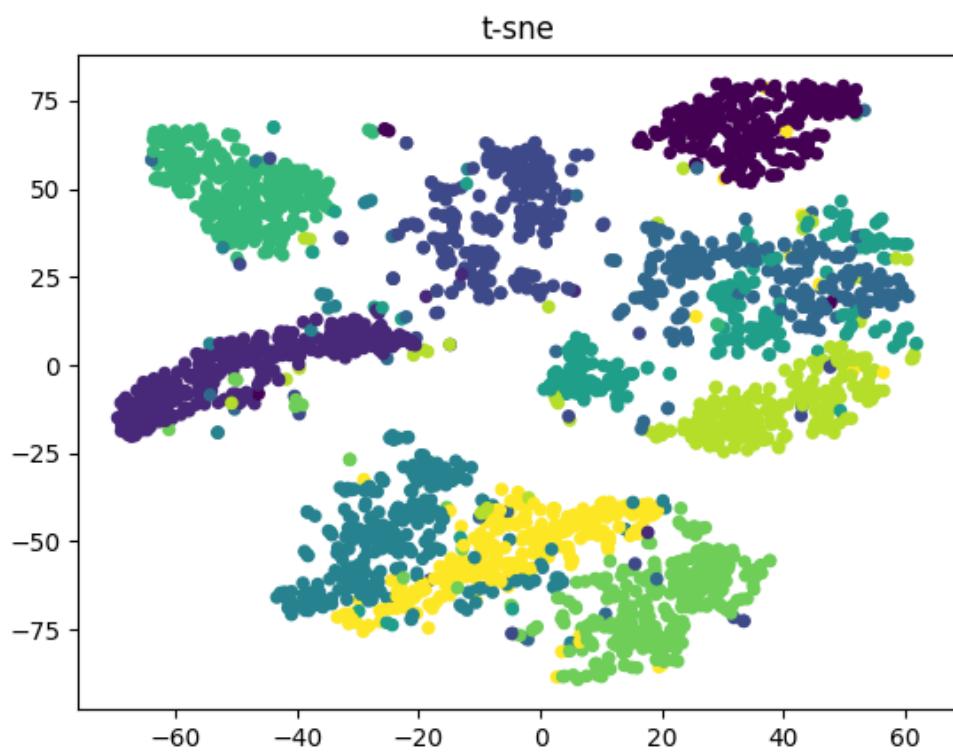
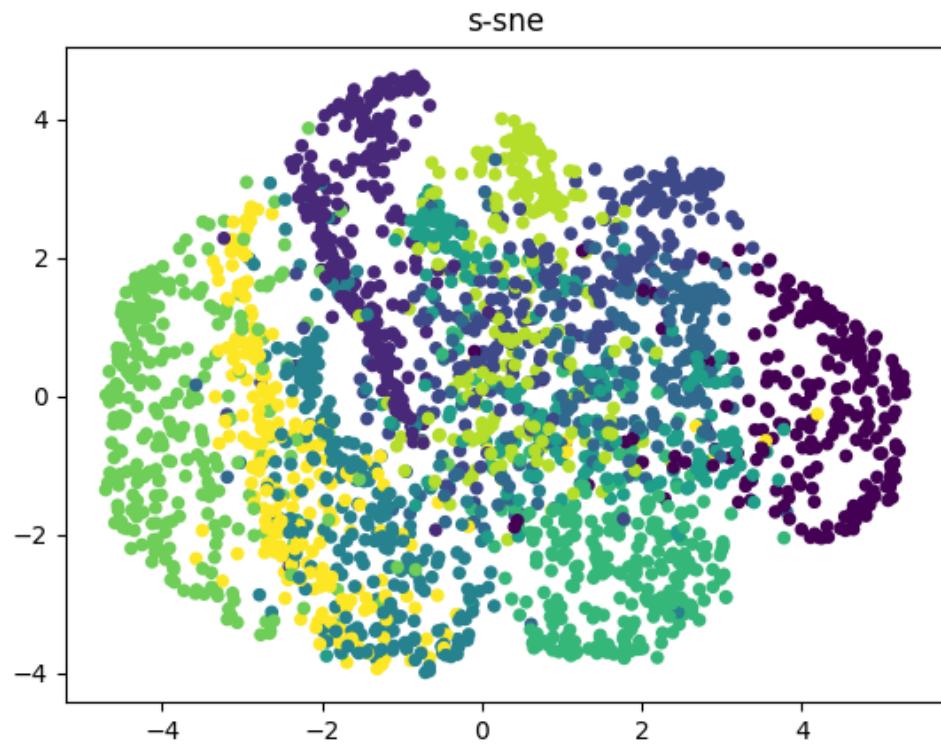
- Perplexity = 25 ([imgur](#), [imgur](#))



- Perplexity = 30 ([imgur](#), [imgur](#))



- Perplexity = 35 ([imgur](#), [imgur](#))



Observations and discussion

Meaning of eigenface

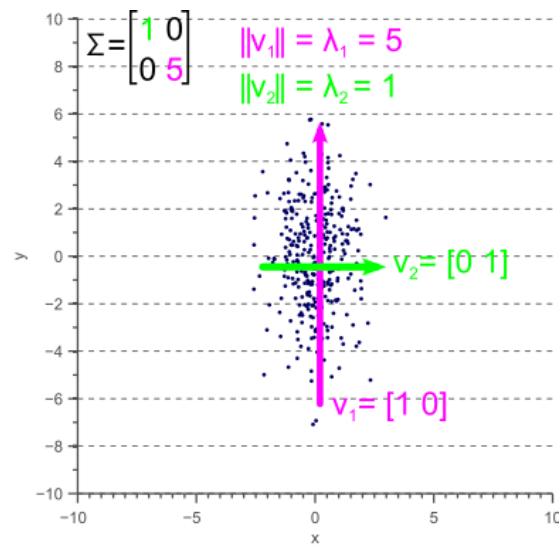
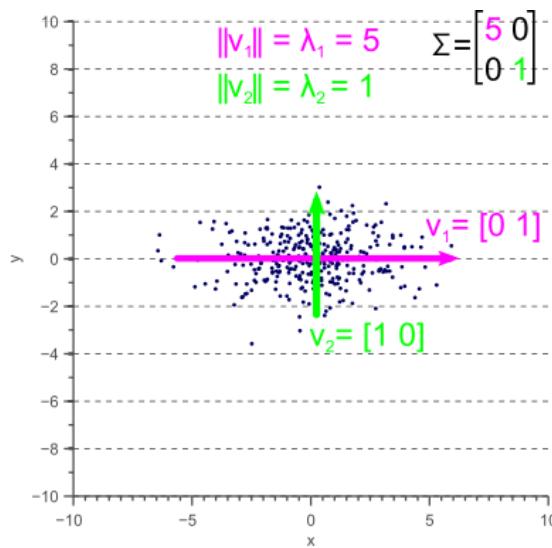
The eigenfaces (eigenvectors) is extracted from eigen decomposition of the covariance matrix of data X .

In the geometry meaning, this process is to extract the independent vectors (eigenvectors).

The linear combination of these independent vectors can be the original data point.

So, an eigenface represents one of the variance of human face that can be an independent factor.

- Eigenvectors (eigenfaces)



- Eigenvalues versus covariance

