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)

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

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

   $$W^{\top}SW$$

3. Project the data onto low dimensional space

   $$Z = XW$$

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

   $$X = ZW^{T}$$

- LDA

1. Within-class scatter S_W

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

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

2. Between-class scatter S_B

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

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

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

   $$Z = XW$$

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

   $$X = ZW^{T}$$

- `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]

```python
def load_data(
    files: list[str],
) -> tuple[npt.NDArray[np.float64], npt.NDArray[np.int32], tuple[int, int]]:
    CROPED_IMAGE_SIZE = (40, 40)
    files.sort()

    text = " ".join(files)
    number_of_images_per_subjects = text.count("subject01")
    number_of_subjects = len(files) // number_of_images_per_subjects

    labels = np.repeat(list(range(number_of_subjects)), number_of_images_per_subjects)

    images = []
    for file in files:
        image = Image.open(file)
        width, height = image.size

        # center crop
        short_side = min(width, height)
        left = (width - short_side) / 2
        top = (height - short_side) / 2
        right = width - left
        bottom = height - top
        image = image.crop((round(left), round(top), round(right), round(bottom)))

        # resize to 40 x 40
        image = image.resize(CROPED_IMAGE_SIZE, resample=Image.Resampling.BILINEAR)

        images.append(np.array(image, dtype=np.float64).flatten() / 255.0)
        image.close()

    images = np.stack(images, axis=0)
    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)

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

```python
if __name__ == "__main__":
    args = Arguments().parse_args()

    os.makedirs(args.out_dir, exist_ok=True)

    pattern = os.path.join(args.root_dir, TRAIN_FOLDER, "*.pgm")
    files = glob.glob(pattern)

    data, labels, _ = load_data(files)

    if args.kernel != "none":
        # PCA assumes the kernel is centered
        centered_x = kernel(data, data, args.kernel, center=True) / data.shape[0]
        x = kernel(data, data, args.kernel)

        pca_weights = solve_by_pca(centered_x, args.kernel, args.num_components)
        fisher_weights = solve_by_fisher(x, labels, args.kernel, args.num_components)
        np.save(os.path.join(args.out_dir, "kernel_eigen.npy"), pca_weights)
        np.save(os.path.join(args.out_dir, "kernel_fisher.npy"), fisher_weights)
    else:
        x = np.cov(data, rowvar=False, bias=True)

        pca_weights = solve_by_pca(x, args.kernel, args.num_components)
        fisher_weights = solve_by_fisher(data, labels, args.kernel, args.num_components)
        np.save(os.path.join(args.out_dir, "eigen.npy"), pca_weights)
        np.save(os.path.join(args.out_dir, "fisher.npy"), fisher_weights)

```

- `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
>
> $$W^\top SW$$

```python
def solve_by_pca(
    x: npt.NDArray[np.float64], kernel_type: KERNEL_TYPES, num_components: int
) -> npt.NDArray[np.float64]:
    # x: D by D matrix (normal)
    # x: N by N matrix (kernel)

    if num_components > x.shape[0]:
        text = "the number of features"
        if kernel_type != "none":
            text = "the number of samples"
        raise ValueError(f"The num_components should not be larger than {text}.")

    start_time = timeit.default_timer()

    eigenvalues, eigenvectors = np.linalg.eigh(x)

    print(
        "Finished solving eigenvectors in",
        timeit.default_timer() - start_time,
        "seconds",
    )

    # find num_components largest eigenvectors
    indices = np.argsort(eigenvalues)[: -num_components - 1 : -1]
    eigenvectors = eigenvectors[:, indices]
    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
>
> $$\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in C_j} x_i$$
>
> $$S_W = \sum_{j=1}^{k} \sum_{i \in C_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^T$$
>
> 2. Between-class scatter S_B
>
> $$\mathbf{m} = \frac{1}{n} \sum x$$
>
> $$S_B = \sum_{j=1}^{k} (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^T$$

```python
def calculate_between_and_with_class_covariance(
    x: npt.NDArray[np.float64], labels: npt.NDArray[np.int32], kernel_type: KERNEL_TYPES
) -> tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]]:
    # calculate global mean: normal + kernel
    global_mean: npt.NDArray[np.float64] = np.mean(x, axis=0, keepdims=True)

    # calculate with-class & between-class covariance
    groups, counts = np.unique(labels, return_counts=True)
    covariance_between_class = np.zeros((x.shape[1],) * 2, dtype=np.float64)
    covariance_within_class = np.zeros_like(covariance_between_class)
    for i in groups:
        group_x = x[labels == i]
        group_mean: npt.NDArray[np.float64] = np.mean(group_x, axis=0, keepdims=True)

        # between-class covariance: normal + kernel
        centered_group_mean = group_mean - global_mean
        covariance_between_class += (
            counts[i] * centered_group_mean.T @ centered_group_mean
        )

        if kernel_type == "none":
            # within-class covariance: normal
            centered_group_x = group_x - group_mean
            covariance_within_class += np.dot(centered_group_x.T, centered_group_x)
        else:
            # within-class covariance: kernel
            group_identity = np.identity(counts[i])
            covariance_within_class += (
                group_x.T
                @ (group_identity - np.full_like(group_identity, 1 / counts[i]))
                @ group_x
            )

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

```
def solve_by_fisher(
    x: npt.NDArray[np.float64],
    labels: npt.NDArray[np.int32],
    kernel_type: KERNEL_TYPES,
    num_components: int,
) → npt.NDArray[np.float64]:
    # x: N by D matrix
    # labels: N array

    if num_components > x.shape[1]:
        text = "the number of features"
        if kernel_type ≠ "none":
            text = "the number of samples"
        raise ValueError(f"The num_components should not be larger than {text}.")

    start_time = timeit.default_timer()
    (
        covariance_between_class,
        covariance_within_class,
    ) = calculate_between_and_with_class_covariance(x, labels, kernel_type)

    print(
        "Finished calculating covariances in",
        timeit.default_timer() - start_time,
        "seconds",
    )

    start_time = timeit.default_timer()

    # S_w^-1 * S_b
    inversed_covariance_within_class = np.linalg.pinv(covariance_within_class)
    eigenvalues, eigenvectors = np.linalg.eigh(
        inversed_covariance_within_class @ covariance_between_class
    )

    print(
        "Finished solving eigenvectors in",
        timeit.default_timer() - start_time,
        "seconds",
    )

    # find num_components largest eigenvectors
    indices = np.argsort(eigenvalues)[: -num_components - 1 : -1]
    eigenvectors = eigenvectors[:, indices]
    return eigenvectors
```

- `test.py` main: program entry point

```python
if __name__ == "__main__":
    args = Arguments().parse_args()

    # load data
    train_data, train_labels, image_size = load_dataset(args.root_dir, TRAIN_FOLDER)
    test_data, test_labels, _ = load_dataset(args.root_dir, TEST_FOLDER)

    # load weights
    pca_weights = np.load(args.eigen_path)
    fisher_weights = np.load(args.fisher_path)

    if args.kernel == "none":
        visualize_faces(
            args.out_dir,
            train_data,
            pca_weights,
            image_size,
            num_eigenfaces=args.num_eigenfaces,
            num_reconstructed_faces=args.num_reconstructed_faces,
            file_name="eigenfaces.jpg",
        )
        visualize_faces(
            args.out_dir,
            train_data,
            fisher_weights,
            image_size,
            num_eigenfaces=args.num_eigenfaces,
            num_reconstructed_faces=args.num_reconstructed_faces,
            file_name="fisherfaces.jpg",
        )
        evaluate(
            train_data,
            train_labels,
            test_data,
            test_labels,
            pca_weights,
            fisher_weights,
            args.k_neighbors,
        )
    else:
        evaluate_kernel(
            train_data,
            train_labels,
            test_data,
            test_labels,
            pca_weights,
            fisher_weights,
            args.k_neighbors,
            args.kernel,
        )

```

- `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

```python
def plot_faces(file_name: str, samples: npt.NDArray[np.float64], cols_per_row: int):
    fig = plt.figure()
    grid = ImageGrid(
        fig,
        111,
        nrows_ncols=(math.ceil(samples.shape[0] / cols_per_row), cols_per_row),
        axes_pad=0,
    )

    for i, ax in enumerate(grid):
        ax: plt.Axes
        ax.imshow(samples[i], cmap="gray")
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)

    fig.savefig(file_name)
    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 $X = XWW^T$.

```python
def visualize_faces(
    out_dir: str,
    samples: npt.NDArray[np.float64],
    weights: npt.NDArray[np.float64],
    image_size: tuple[int, int],
    file_name: str = "eigenfaces.jpg",
    num_eigenfaces: int = 25,
    num_reconstructed_faces: int = 10,
    cols_per_row: int = 5,
):
    # D: features
    # E: num_components
    # samples: N by D matrix
    # weights: D by E matrix
    width, height = image_size
    os.makedirs(out_dir, exist_ok=True)

    weights = weights[:, :num_eigenfaces]
    # visualize eigenfaces
    plot_faces(
        os.path.join(out_dir, file_name),
        weights.T.reshape(num_eigenfaces, width, height),
        cols_per_row,
    )

    # randomly sample reconstructed_faces
    indices = np.random.choice(samples.shape[0], num_reconstructed_faces, replace=False)
    samples = samples[indices]

    # visualize original faces
    plot_faces(
        os.path.join(out_dir, f"original_reconstructed_{file_name}"),
        samples.reshape(num_reconstructed_faces, width, height),
        cols_per_row,
    )

    # reconstruct faces
    faces = samples @ weights @ weights.T

    # visualize reconstructed faces
    plot_faces(
        os.path.join(out_dir, f"reconstructed_{file_name}"),
        faces.reshape(num_reconstructed_faces, width, height),
        cols_per_row,
    )

    error = cdist(faces, samples, metric="euclidean").diagonal().mean()
    print("Reconstruction Average Error: ", error)
```

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

```python
def evaluate(
    train_data: npt.NDArray[np.float64],
    train_labels: npt.NDArray[np.int32],
    test_data: npt.NDArray[np.float64],
    test_labels: npt.NDArray[np.int32],
    pca_weights: npt.NDArray[np.float64],
    fisher_weights: npt.NDArray[np.float64],
    k_neighbors: int,
):
    pca_predictions = classify(
        train_data, train_labels, test_data, pca_weights, k_neighbors
    )
    fisher_predictions = classify(
        train_data, train_labels, test_data, fisher_weights, k_neighbors
    )

    print("K-NN Error rate (PCA):", np.mean(pca_predictions ≠ test_labels))
    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
>
> To project the data onto low dimensional space,
>
> $Z = XW$

```python
def classify(
    train_data: npt.NDArray[np.float64],
    train_labels: npt.NDArray[np.int32],
    test_data: npt.NDArray[np.float64],
    weights: npt.NDArray[np.float64],
    k_neighbors: int = 5,
) -> npt.NDArray[np.int32]:
    # project onto low dimensional space
    train_data = train_data @ weights
    test_data = test_data @ weights

    # find labels of k neighbors
    distance = cdist(test_data, train_data, metric="euclidean")
    k_indices = np.argsort(distance, axis=-1)[:, :k_neighbors]
    labels = train_labels[k_indices.flatten()].reshape(-1, k_neighbors)

    # find mode of labels
    predictions = []
    for row in labels:
        classes, counts = np.unique(row, return_counts=True)
        predictions.append(classes[np.argmax(counts)])

    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

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

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

       $\mathbf{S}_W = \sum_{j=1}^{k} K_j (I - 1_{n_j}) K_j^T$
    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.
>
> $K^C = K - 1_N K - K 1_N + 1_N K 1_N$

```python
def kernel(
    x: npt.NDArray[np.float64],
    y: npt.NDArray[np.float64],
    kernel_type: KERNEL_TYPES,
    center: bool = False,
) -> npt.NDArray[np.float64]:
    gamma = 1 / x.shape[1]
    if kernel_type == "rbf":
        kernel = np.exp(-gamma * cdist(x, y, metric="sqeuclidean"))
    else:
        kernel = x @ y.T
        if kernel_type == "poly":
            kernel = (gamma * kernel) ** 3

    if center:
        # center the matrix
        # K_C = K - 1_N * K - K * 1_N + 1_N * K * 1_N
        all_inv_n_matrix = np.full_like(kernel, 1 / kernel.shape[0])
        kernel = (
            kernel
            - all_inv_n_matrix @ kernel
            - kernel @ all_inv_n_matrix
            + all_inv_n_matrix @ kernel @ all_inv_n_matrix
        )

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

```python
if __name__ == "__main__":
    args = Arguments().parse_args()

    os.makedirs(args.out_dir, exist_ok=True)

    pattern = os.path.join(args.root_dir, TRAIN_FOLDER, "*.pgm")
    files = glob.glob(pattern)

    data, labels, _ = load_data(files)

    if args.kernel != "none":
        # PCA assumes the kernel is centered
        centered_x = kernel(data, data, args.kernel, center=True) / data.shape[0]
        x = kernel(data, data, args.kernel)

        pca_weights = solve_by_pca(centered_x, args.kernel, args.num_components)
        fisher_weights = solve_by_fisher(x, labels, args.kernel, args.num_components)
        np.save(os.path.join(args.out_dir, "kernel_eigen.npy"), pca_weights)
        np.save(os.path.join(args.out_dir, "kernel_fisher.npy"), fisher_weights)
    else:
        x = np.cov(data, rowvar=False, bias=True)

        pca_weights = solve_by_pca(x, args.kernel, args.num_components)
        fisher_weights = solve_by_fisher(data, labels, args.kernel, args.num_components)
        np.save(os.path.join(args.out_dir, "eigen.npy"), pca_weights)
        np.save(os.path.join(args.out_dir, "fisher.npy"), fisher_weights)

```

- `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
>
> $$Ka = \lambda Na$$
>
> $$\frac{1}{N}Ka = \lambda a$$

```python
def solve_by_pca(
    x: npt.NDArray[np.float64], kernel_type: KERNEL_TYPES, num_components: int
) -> npt.NDArray[np.float64]:
    # x: D by D matrix (normal)
    # x: N by N matrix (kernel)

    if num_components > x.shape[0]:
        text = "the number of features"
        if kernel_type != "none":
            text = "the number of samples"
        raise ValueError(f"The num_components should not be larger than {text}.")

    start_time = timeit.default_timer()

    eigenvalues, eigenvectors = np.linalg.eigh(x)

    print(
        "Finished solving eigenvectors in",
        timeit.default_timer() - start_time,
        "seconds",
    )

    # find num_components largest eigenvectors
    indices = np.argsort(eigenvalues)[: -num_components - 1 : -1]
    eigenvectors = eigenvectors[:, indices]
    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 $\mathbf{S}_W$
>
> $$\mathbf{M}_j = \frac{1}{N_j} \sum_{k \in C_j} k(x_k, x_i), i \in N$$
>
> $$\mathbf{S}_W = \sum_{j=1}^{k} K_j (I - 1_{n_j}) K_j^T$$
>
> 2. Between-class scatter $\mathbf{S}_B$
>
> $$\mathbf{M} = \frac{1}{N} \sum_{k \in N} k(x_k, x_i), i \in N$$
>
> $$\mathbf{S}_B = \sum_{j=1}^{k} N_j (\mathbf{M}_j - \mathbf{M})(\mathbf{M}_j - \mathbf{M})^T$$

```python
def calculate_between_and_with_class_covariance(
    x: npt.NDArray[np.float64], labels: npt.NDArray[np.int32], kernel_type: KERNEL_TYPES
) -> tuple[npt.NDArray[np.float64], npt.NDArray[np.float64]]:
    # calculate global mean: normal + kernel
    global_mean: npt.NDArray[np.float64] = np.mean(x, axis=0, keepdims=True)

    # calculate with-class & between-class covariance
    groups, counts = np.unique(labels, return_counts=True)
    covariance_between_class = np.zeros((x.shape[1],) * 2, dtype=np.float64)
    covariance_within_class = np.zeros_like(covariance_between_class)
    for i in groups:
        group_x = x[labels == i]
        group_mean: npt.NDArray[np.float64] = np.mean(group_x, axis=0, keepdims=True)

        # between-class covariance: normal + kernel
        centered_group_mean = group_mean - global_mean
        covariance_between_class += (
            counts[i] * centered_group_mean.T @ centered_group_mean
        )

        if kernel_type == "none":
            # within-class covariance: normal
            centered_group_x = group_x - group_mean
            covariance_within_class += np.dot(centered_group_x.T, centered_group_x)
        else:
            # within-class covariance: kernel
            group_identity = np.identity(counts[i])
            covariance_within_class += (
                group_x.T
                @ (group_identity - np.full_like(group_identity, 1 / counts[i]))
                @ group_x
            )

    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.

```
def solve_by_fisher(
    x: npt.NDArray[np.float64],
    labels: npt.NDArray[np.int32],
    kernel_type: KERNEL_TYPES,
    num_components: int,
) → npt.NDArray[np.float64]:
    # x: N by D matrix
    # labels: N array

    if num_components > x.shape[1]:
        text = "the number of features"
        if kernel_type ≠ "none":
            text = "the number of samples"
        raise ValueError(f"The num_components should not be larger than {text}.")

    start_time = timeit.default_timer()
    (
        covariance_between_class,
        covariance_within_class,
    ) = calculate_between_and_with_class_covariance(x, labels, kernel_type)

    print(
        "Finished calculating covariances in",
        timeit.default_timer() - start_time,
        "seconds",
    )

    start_time = timeit.default_timer()

    # S_w^-1 * S_b
    inversed_covariance_within_class = np.linalg.pinv(covariance_within_class)
    eigenvalues, eigenvectors = np.linalg.eigh(
        inversed_covariance_within_class @ covariance_between_class
    )

    print(
        "Finished solving eigenvectors in",
        timeit.default_timer() - start_time,
        "seconds",
    )

    # find num_components largest eigenvectors
    indices = np.argsort(eigenvalues)[: -num_components - 1 : -1]
    eigenvectors = eigenvectors[:, indices]
    return eigenvectors
```

- `test.py` main: program entry point

```python
if __name__ == "__main__":
    args = Arguments().parse_args()

    # load data
    train_data, train_labels, image_size = load_dataset(args.root_dir, TRAIN_FOLDER)
    test_data, test_labels, _ = load_dataset(args.root_dir, TEST_FOLDER)

    # load weights
    pca_weights = np.load(args.eigen_path)
    fisher_weights = np.load(args.fisher_path)

    if args.kernel == "none":
        visualize_faces(
            args.out_dir,
            train_data,
            pca_weights,
            image_size,
            num_eigenfaces=args.num_eigenfaces,
            num_reconstructed_faces=args.num_reconstructed_faces,
            file_name="eigenfaces.jpg",
        )
        visualize_faces(
            args.out_dir,
            train_data,
            fisher_weights,
            image_size,
            num_eigenfaces=args.num_eigenfaces,
            num_reconstructed_faces=args.num_reconstructed_faces,
            file_name="fisherfaces.jpg",
        )
        evaluate(
            train_data,
            train_labels,
            test_data,
            test_labels,
            pca_weights,
            fisher_weights,
            args.k_neighbors,
        )
    else:
        evaluate_kernel(
            train_data,
            train_labels,
            test_data,
            test_labels,
            pca_weights,
            fisher_weights,
            args.k_neighbors,
            args.kernel,
        )

```

- `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 $K(X, X)$ and $K(X_{new}, X)$.

```python
def evaluate_kernel(
    train_data: npt.NDArray[np.float64],
    train_labels: npt.NDArray[np.int32],
    test_data: npt.NDArray[np.float64],
    test_labels: npt.NDArray[np.int32],
    pca_weights: npt.NDArray[np.float64],
    fisher_weights: npt.NDArray[np.float64],
    k_neighbors: int,
    kernel_type: KERNEL_TYPES,
):
    # calculate kernels
    train_kernel = kernel(train_data, train_data, kernel_type)
    test_kernel = kernel(test_data, train_data, kernel_type)

    pca_predictions = classify(
        train_kernel,
        train_labels,
        test_kernel,
        pca_weights,
        k_neighbors,
    )
    fisher_predictions = classify(
        train_kernel, train_labels, test_kernel, fisher_weights, k_neighbors
    )
```

- `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
>
> To project the data onto low dimensional space,
>
> $Z = K(X, X)A$

```python
def classify(
    train_data: npt.NDArray[np.float64],
    train_labels: npt.NDArray[np.int32],
    test_data: npt.NDArray[np.float64],
    weights: npt.NDArray[np.float64],
    k_neighbors: int = 5,
) -> npt.NDArray[np.int32]:
    # project onto low dimensional space
    train_data = train_data @ weights
    test_data = test_data @ weights

    # find labels of k neighbors
    distance = cdist(test_data, train_data, metric="euclidean")
    k_indices = np.argsort(distance, axis=-1)[:, :k_neighbors]
    labels = train_labels[k_indices.flatten()].reshape(-1, k_neighbors)

    # find mode of labels
    predictions = []
    for row in labels:
        classes, counts = np.unique(row, return_counts=True)
        predictions.append(classes[np.argmax(counts)])

    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.

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

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

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

    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.

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

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

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

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

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

4. Calculate the gradient for each y_i

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

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

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

```python
def sne(
    X=np.array([]),
    no_dims=2,
    initial_dims=50,
    perplexity=30.0,
    mode: Mode = Mode.symmetric_sne,
):
    """
    Runs t-SNE on the dataset in the NxD array X to reduce its
    dimensionality to no_dims dimensions. The syntaxis of the function is
    `Y = tsne.tsne(X, no_dims, perplexity), where X is an NxD NumPy array.
    """

    # Check inputs
    if not np.issubdtype(X.dtype, np.floating):
        print("Error: array X should be a float array.")
        return -1
    if isinstance(no_dims, float):
        print("Error: number of dimensions should be an integer.")
        return -1

    # Initialize variables
    X = pca(X, initial_dims).real
    (n, d) = X.shape
    max_iter = 1000
    initial_momentum = 0.5
    final_momentum = 0.8
    eta = 500
    min_gain = 0.01
    Y = np.random.randn(n, no_dims)
    dY = np.zeros((n, no_dims))
    iY = np.zeros((n, no_dims))
    gains = np.ones((n, no_dims))
    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**.

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

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

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

```python
def x2p(X=np.array([]), tol=1e-5, perplexity=30.0):
    """
    Performs a binary search to get P-values in such a way that each
    conditional Gaussian has the same perplexity.
    """

    # Initialize some variables
    print("Computing pairwise distances...")
    (n, d) = X.shape
    sum_X = np.sum(np.square(X), 1)
    D = np.add(np.add(-2 * np.dot(X, X.T), sum_X).T, sum_X)
    P = np.zeros((n, n))
    beta = np.ones((n, 1))
    logU = np.log(perplexity)

    # Loop over all datapoints
    for i in range(n):
        # Print progress
        if i % 500 == 0:
            print("Computing P-values for point %d of %d..." % (i, n))

        # Compute the Gaussian kernel and entropy for the current precision
        betamin = -np.inf
        betamax = np.inf
        Di = D[i, np.concatenate((np.r_[0:i], np.r_[i + 1 : n]))]
        (H, thisP) = Hbeta(Di, beta[i])

        # Evaluate whether the perplexity is within tolerance
        Hdiff = H - logU
        tries = 0
        while np.abs(Hdiff) > tol and tries < 50:
            # If not, increase or decrease precision
            if Hdiff > 0:
                betamin = beta[i].copy()
                if betamax == np.inf or betamax == -np.inf:
                    beta[i] = beta[i] * 2.0
                else:
                    beta[i] = (beta[i] + betamax) / 2.0
            else:
                betamax = beta[i].copy()
                if betamin == np.inf or betamin == -np.inf:
                    beta[i] = beta[i] / 2.0
                else:
                    beta[i] = (beta[i] + betamin) / 2.0

            # Recompute the values
            (H, thisP) = Hbeta(Di, beta[i])
            Hdiff = H - logU
            tries += 1

        # Set the final row of P
        P[i, np.concatenate((np.r_[0:i], np.r_[i + 1 : n]))] = thisP

    # Return final P-matrix
    print("Mean value of sigma: %f" % np.mean(np.sqrt(1 / beta)))
    return P
```

- `sne` function: after calculating the conditional probability $p_{i|j}$, 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)
```

- `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

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

5th step

Use optimization algorithm to update y_i

```
    # Run iterations
    for iter in range(max_iter):
        # Compute pairwise affinities
        sum_Y = np.sum(np.square(Y), 1)
        num = -2.0 * np.dot(Y, Y.T)
        if mode == Mode.symmetric_sne:
            num = np.exp(-np.add(np.add(num, sum_Y).T, sum_Y))
        else:
            num = 1.0 / (1.0 + np.add(np.add(num, sum_Y).T, sum_Y))
        num[range(n), range(n)] = 0.0
        Q = num / np.sum(num)
        Q = np.maximum(Q, 1e-12)

        # Compute gradient
        PQ = P - Q
        for i in range(n):
            if mode == Mode.symmetric_sne:
                dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
            else:
                dY[i, :] = np.sum(
                    np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0
                )

        # Perform the update
        if iter < 20:
            momentum = initial_momentum
        else:
            momentum = final_momentum
        gains = (gains + 0.2) * ((dY > 0.0) != (iY > 0.0)) + (gains * 0.8) * (
            (dY > 0.0) == (iY > 0.0)
        )
        gains[gains < min_gain] = min_gain
        iY = momentum * iY - eta * (gains * dY)
        Y = Y + iY
        Y = Y - np.tile(np.mean(Y, 0), (n, 1))

        # 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))

        # Stop lying about P-values
        if iter == 100:
            P = P / 4.0
```

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

```python
def draw_results(
    step: int, Y: np.ndarray, labels: np.ndarray, perplexity: float, mode: Mode
):
    fig = pylab.figure()
    ax1 = fig.subplots(1, 1)

    fig.suptitle(f"Step = {step}, Algorithm = {str(mode)}, Perplexity = {perplexity}")

    ax1.scatter(Y[:, 0], Y[:, 1], 20, labels)

    fig.canvas.draw()
    image = Image.frombytes(
        "RGB", fig.canvas.get_width_height(), fig.canvas.tostring_rgb()
    )

    pylab.close(fig)
    return image
```

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

```python
        # Stop lying about P-values
        if iter == 100:
            P = P / 4.0

    history_images[0].save(
        os.path.join(OUT_DIR, f"{str(mode)}_history.gif"),
        save_all=True,
        append_images=history_images[1:],
        optimize=False,
        loop=0,
        duration=300,
    )
    visualize_similarities(P, Q, labels, perplexity, mode)

    # Return solution
    return Y
```

Part3

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

```python
        # Stop lying about P-values
        if iter == 100:
            P = P / 4.0

    history_images[0].save(
        os.path.join(OUT_DIR, f"{str(mode)}_history.gif"),
        save_all=True,
        append_images=history_images[1:],
        optimize=False,
        loop=0,
        duration=300,
    )
    visualize_similarities(P, Q, labels, perplexity, mode)

    # Return solution
    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.

```python
def visualize_similarities(
    P: np.ndarray, Q: np.ndarray, labels: np.ndarray, perplexity: float, mode: Mode
):
    # reorder rows and cols by labels
    # block-wise similarity matrix
    indices = np.argsort(labels)
    P = P[indices][:, indices]
    Q = Q[indices][:, indices]

    min_P = np.min(P)
    scaled_P = (P - min_P) / (np.max(P) - min_P)

    min_Q = np.min(Q)
    scaled_Q = (Q - min_Q) / (np.max(Q) - min_Q)

    fig = pylab.figure()
    ax1, ax2 = fig.subplots(1, 2)

    fig.suptitle(f"Algorithm = {str(mode)}, Perplexity = {perplexity}")

    ax1.set_title("High-dimensional space")
    image = ax1.matshow(scaled_P, cmap="hot")
    ax1.set_xlabel("label")
    ax1.set_ylabel("label")
    fig.colorbar(image)

    ax2.set_title("Low-dimensional space")
    image = ax2.matshow(scaled_Q, cmap="hot")
    ax2.set_xlabel("label")
    ax2.set_ylabel("label")
    fig.colorbar(image)

    fig.tight_layout()
    fig.savefig(os.path.join(OUT_DIR, f"{str(mode)}_similarities.png"))

    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.

```python
class Mode(Enum):
    symmetric_sne = "s-sne"
    t_sne = "t-sne"

    def __str__(self):
        return self.value
```

```python
if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        formatter_class=argparse.ArgumentDefaultsHelpFormatter
    )
    parser.add_argument(
        "--algorithm",
        type=Mode,
        choices=list(Mode),
        default=Mode.symmetric_sne,
        help="Choose an algorithm to perform dimensional rediction",
    )
    parser.add_argument(
        "--perplexity",
        type=float,
        default=20.0,
        help="perplexity",
    )
    args = parser.parse_args()

    print("Running example on 2,500 MNIST digits ...")
    X = np.loadtxt("data/mnist2500_X.txt")
    labels = np.loadtxt("data/mnist2500_labels.txt")

    os.makedirs(OUT_DIR, exist_ok=True)

    Y = sne(
        X, no_dims=2, initial_dims=50, perplexity=args.perplexity, mode=args.algorithm
    )
    pylab.title(str(args.algorithm))
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(os.path.join(OUT_DIR, f"{str(args.algorithm)}_final.png"))
    pylab.close()
```

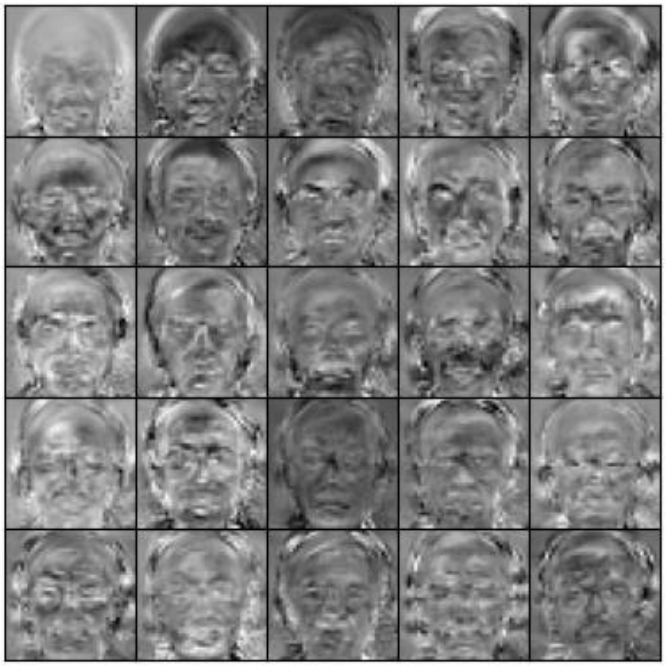
Experiments settings and results & Discussion

Kernel Eigenfaces

Part1

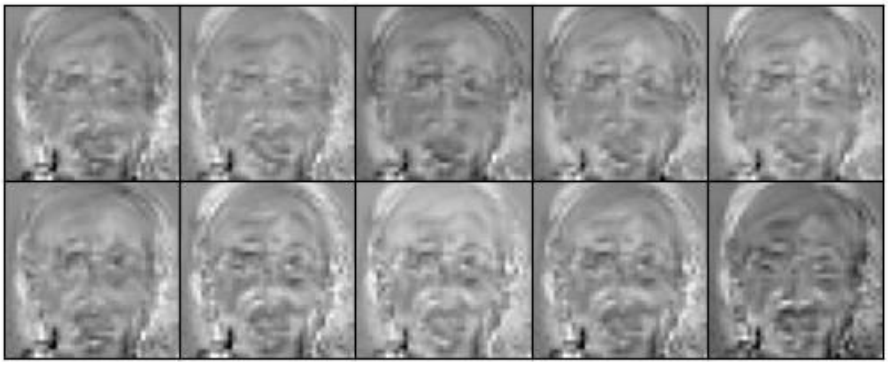
- Eigenfaces

- 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 kerel.
- 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): 0.26666666666666666
```

t-SNE

Part1

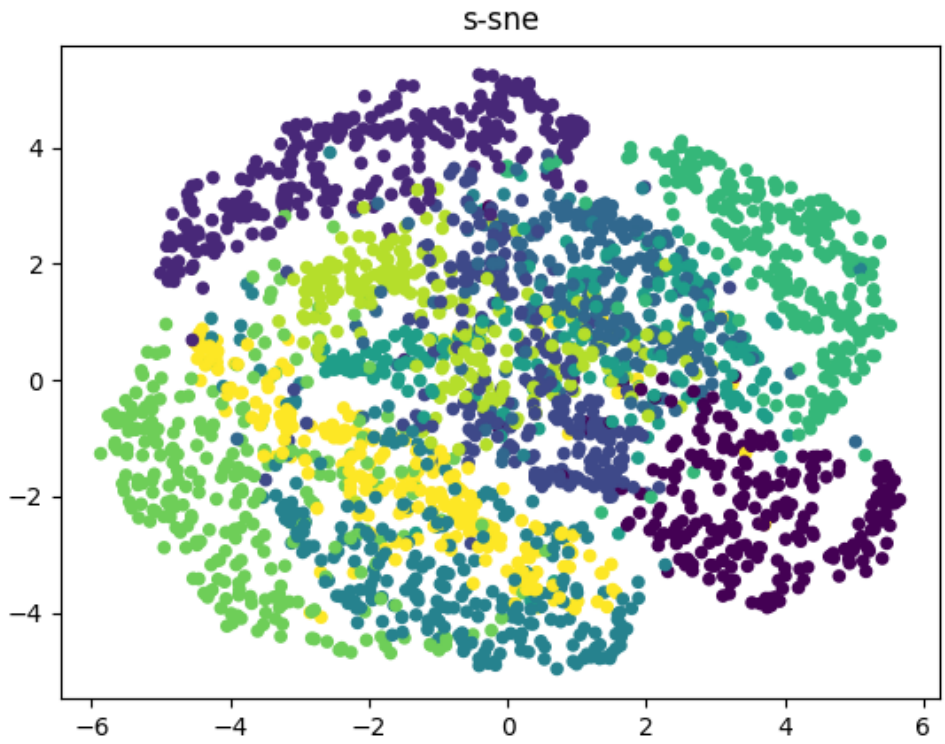
Observations

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).

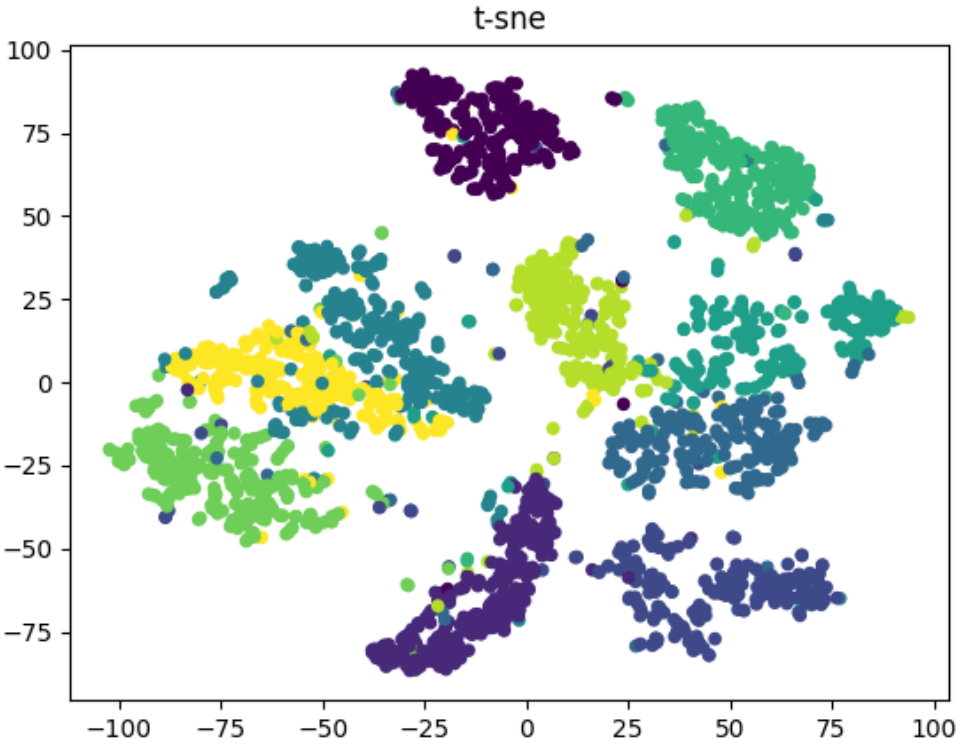
So, in the low dimensional space, it proves that t-SNE uses student t-distribution to formulate the data points better than s-SNE's.

Results

- Symmetric SNE

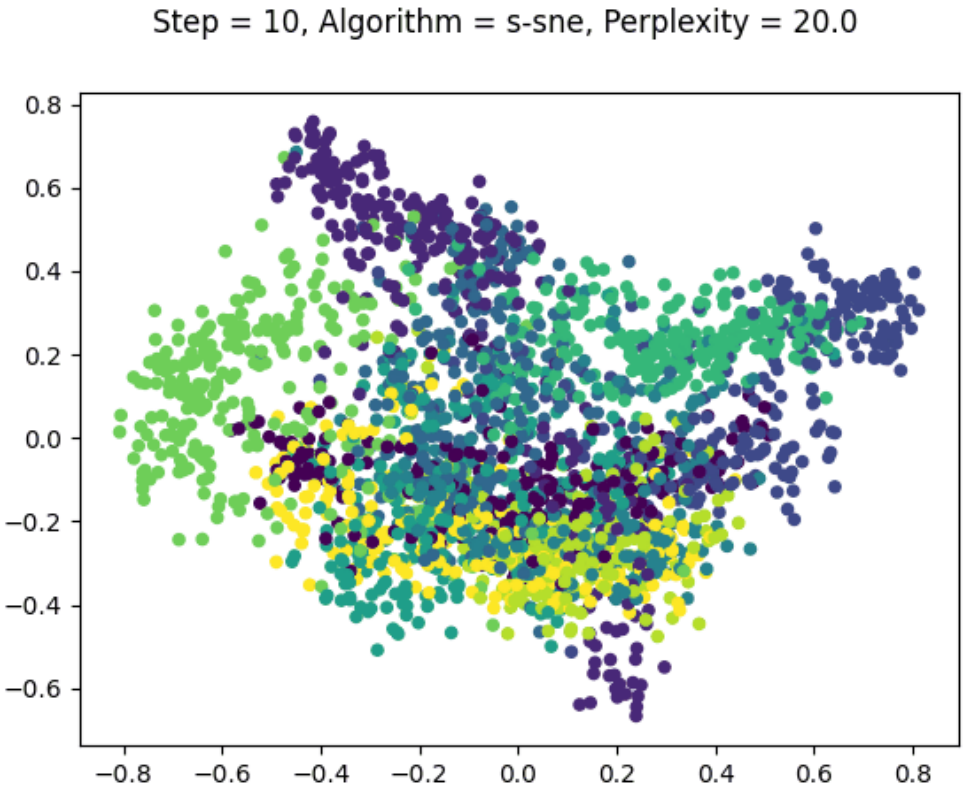


- t-SNE

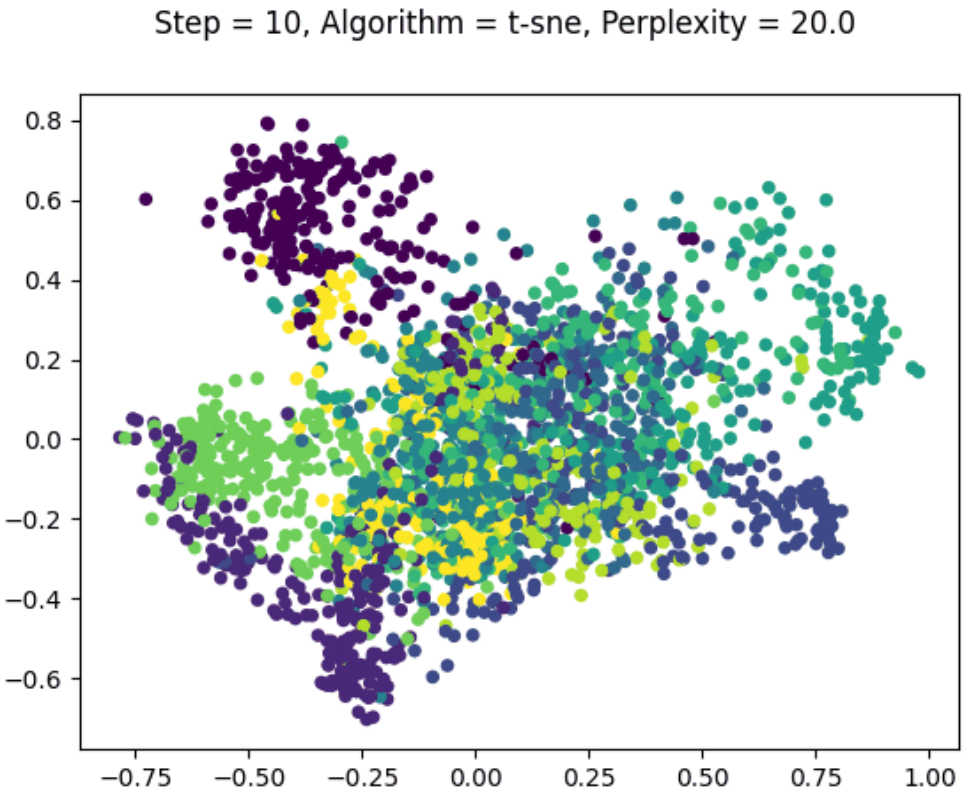


Part2

- s-SNE (imgur gif)

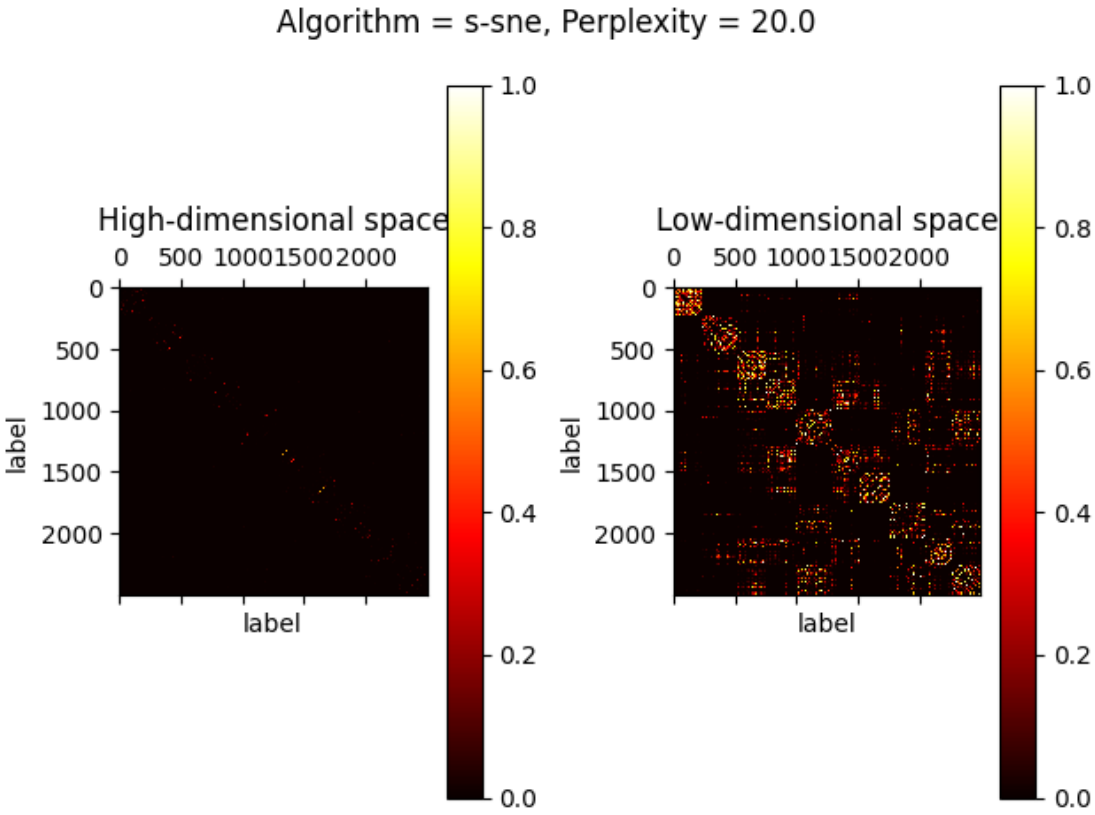


- t-SNE (imgur gif)

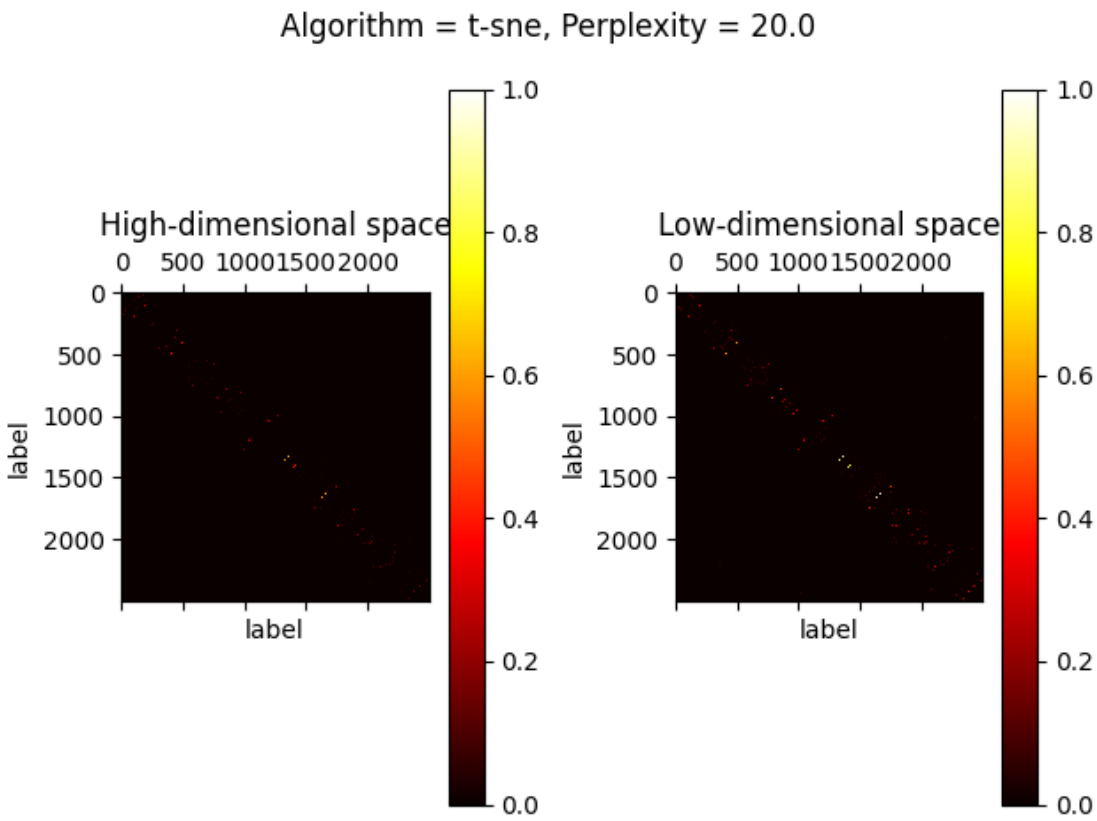


Part3

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



- t-SNE



Part4

Observations

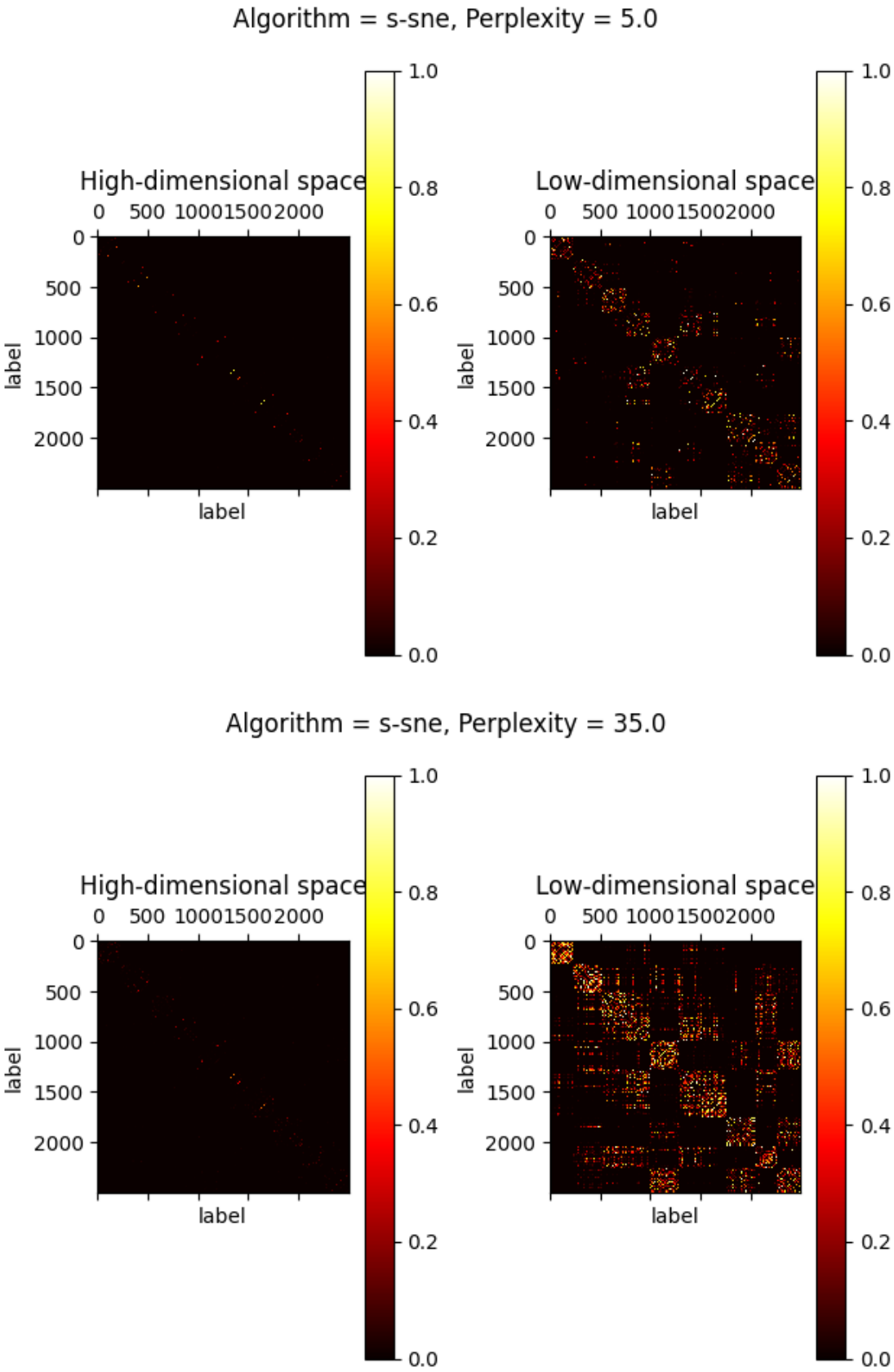
- In these points figures, we could know
  - when the perplexity is low (entropy is low), the variance of each group is very small.
  - when the perplexity is high (entropy is high), the variance of each group is very wide.
  - It proves that the perplexity will affect the choose of variance.
- The t-SNE 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.

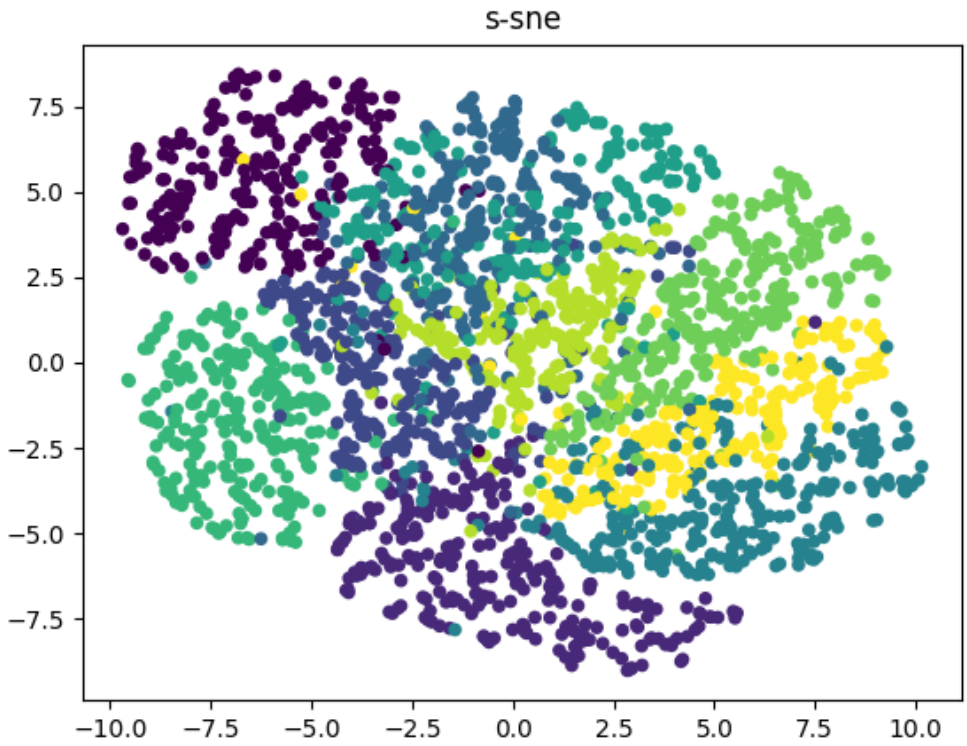
  Look at the low dimensional space, because of the crowded problem, we could see the data points are very close.

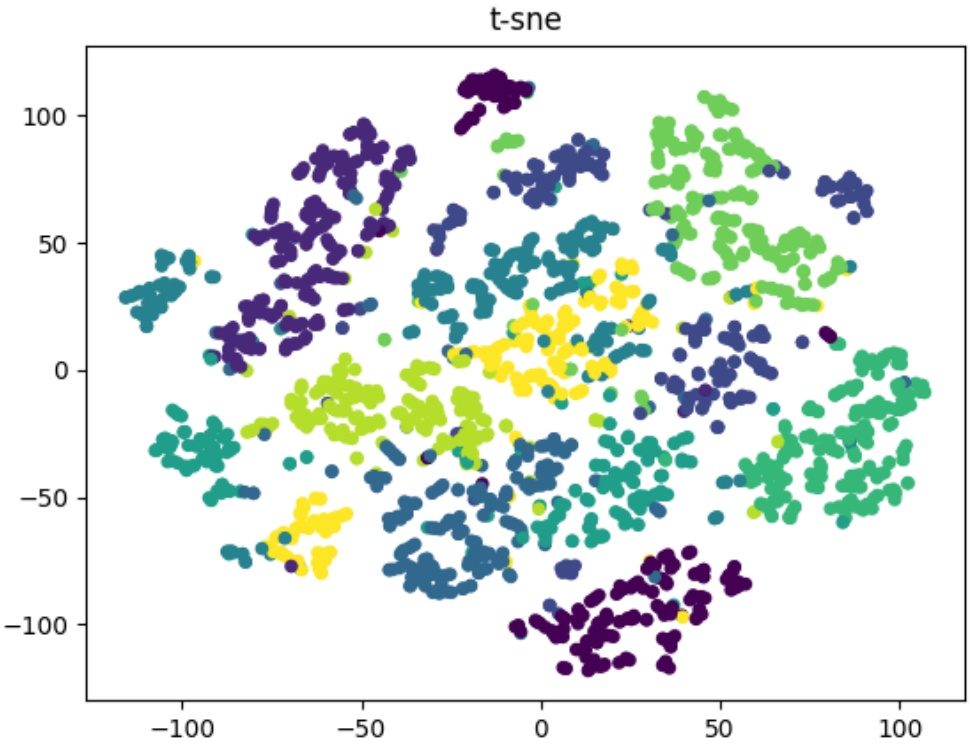
  When the perplexity is going higher, the variance is wider. It is very obvious that the crowded problem is going worse.



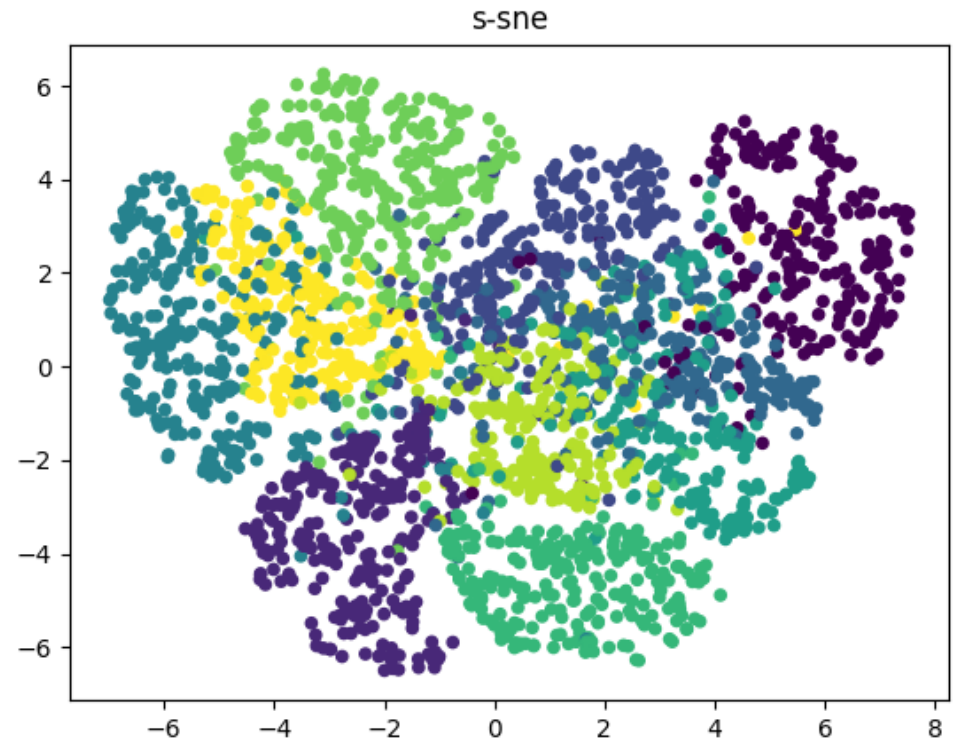
Results

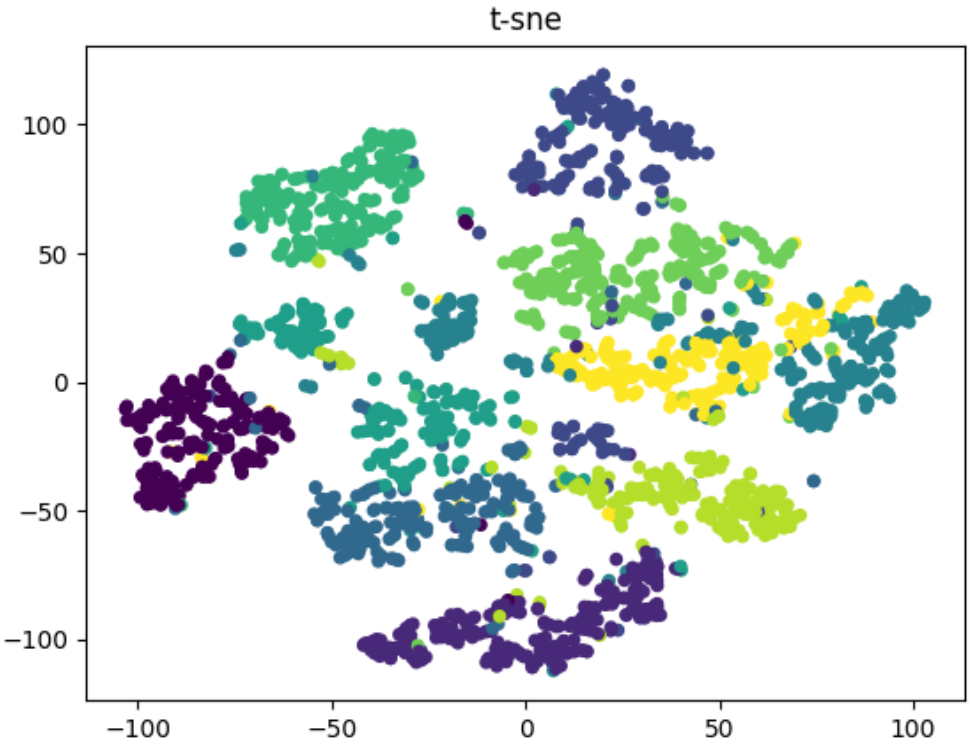
- Perplexity = 5 (imgur, imgur)



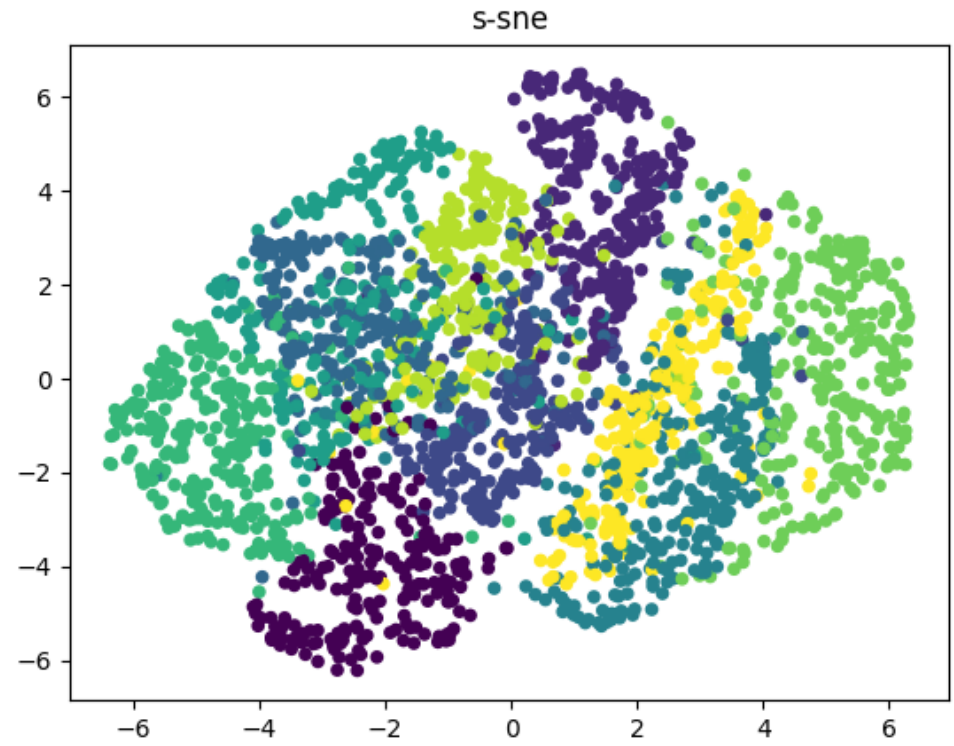


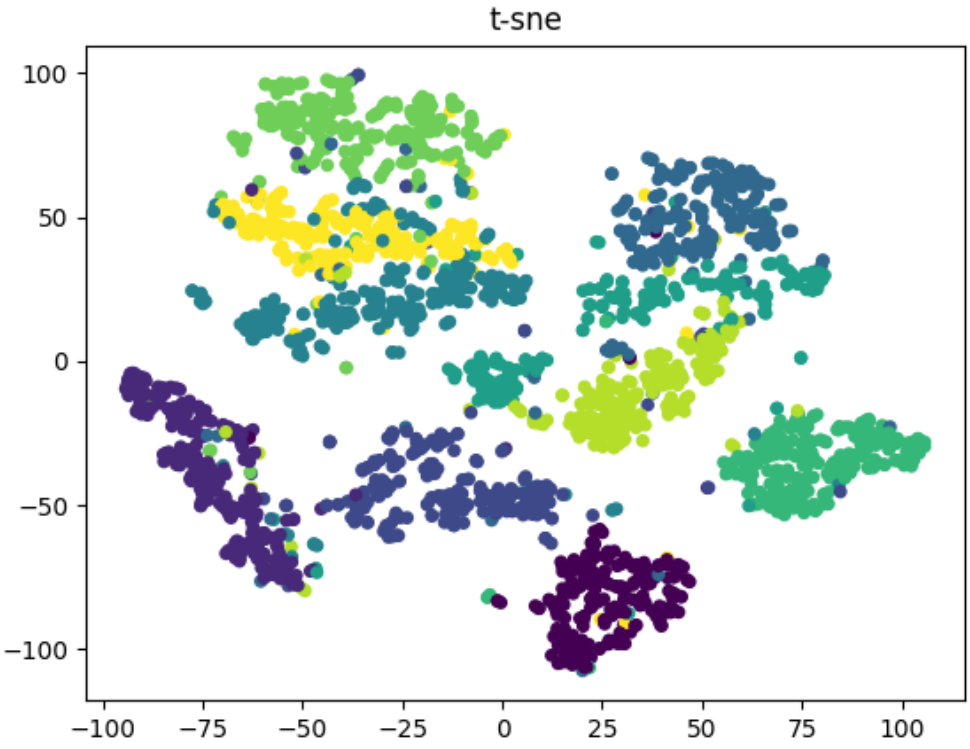
- Perplexity = 10 (imgur, imgur)



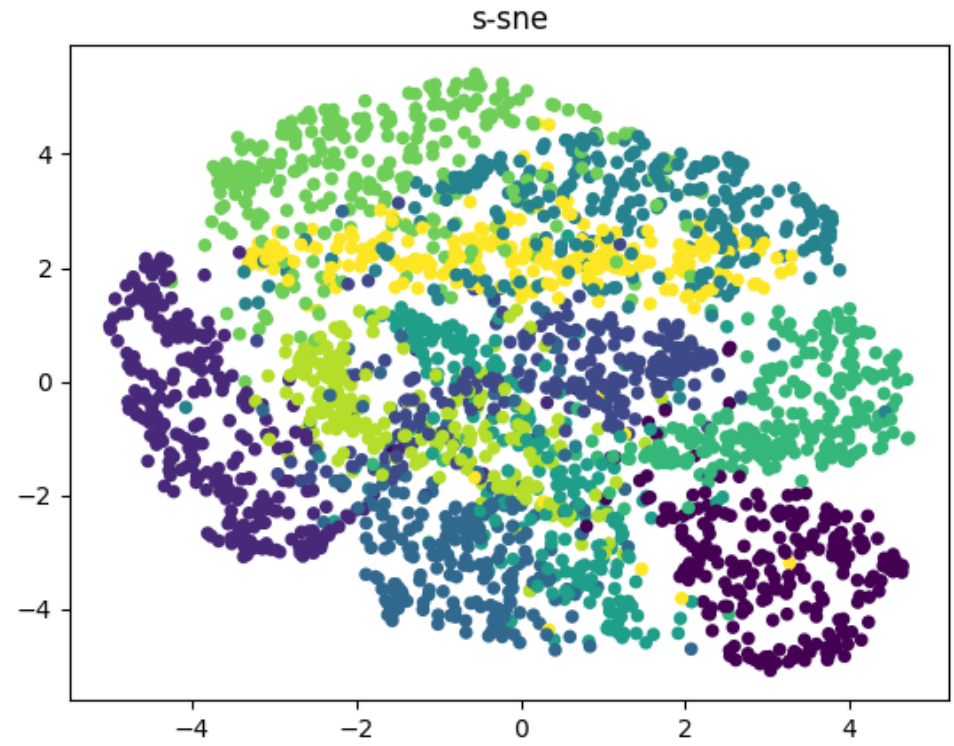


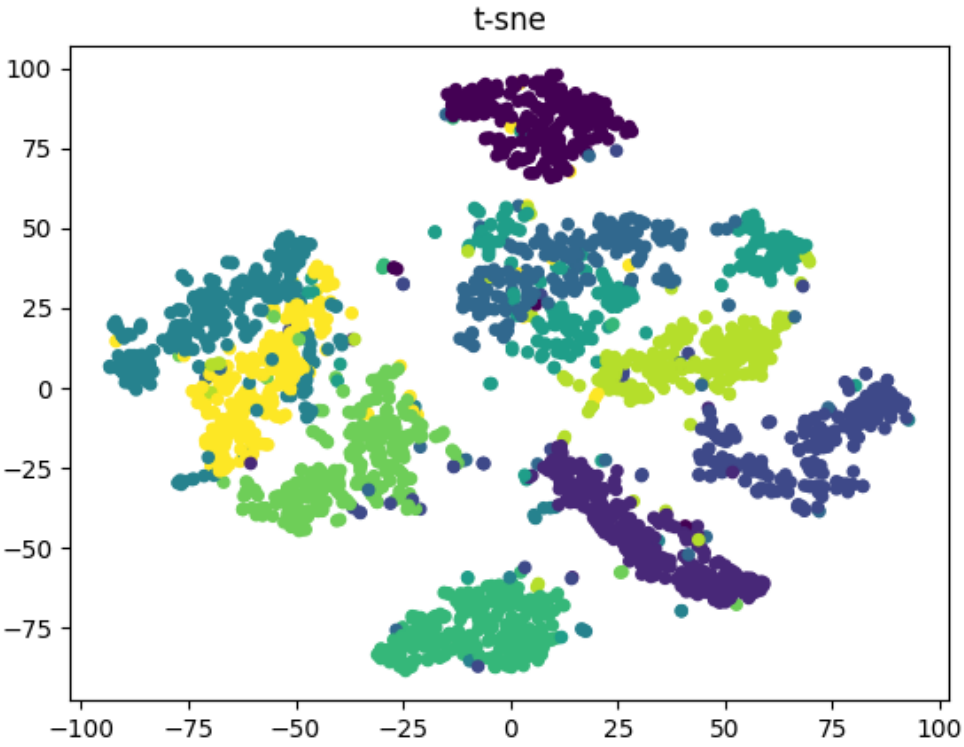
- Perplexity = 15 (imgur, imgur)



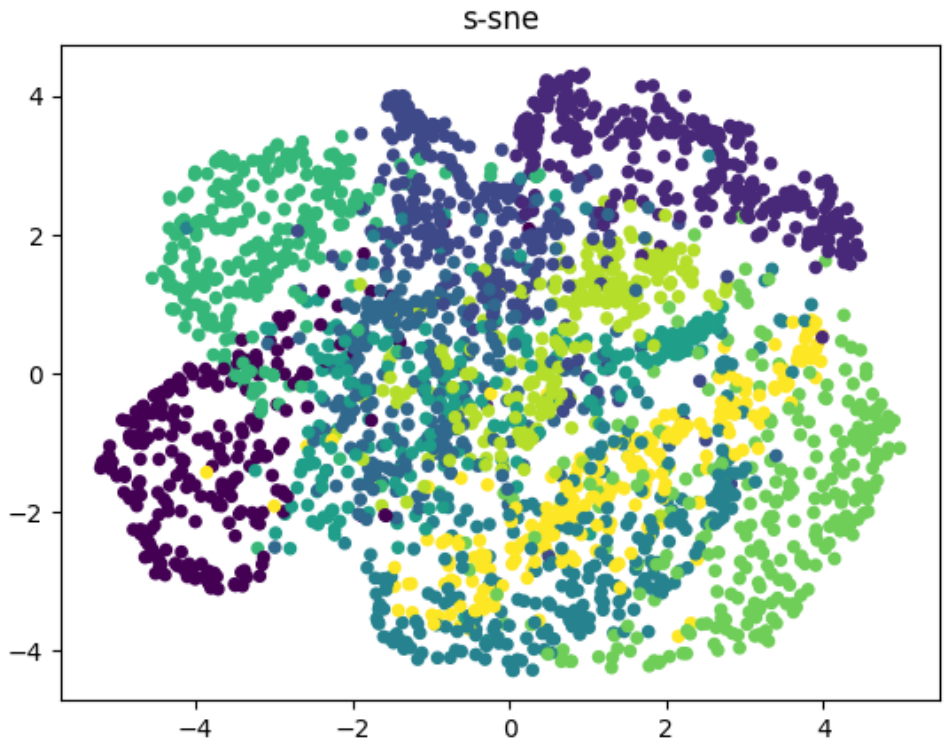


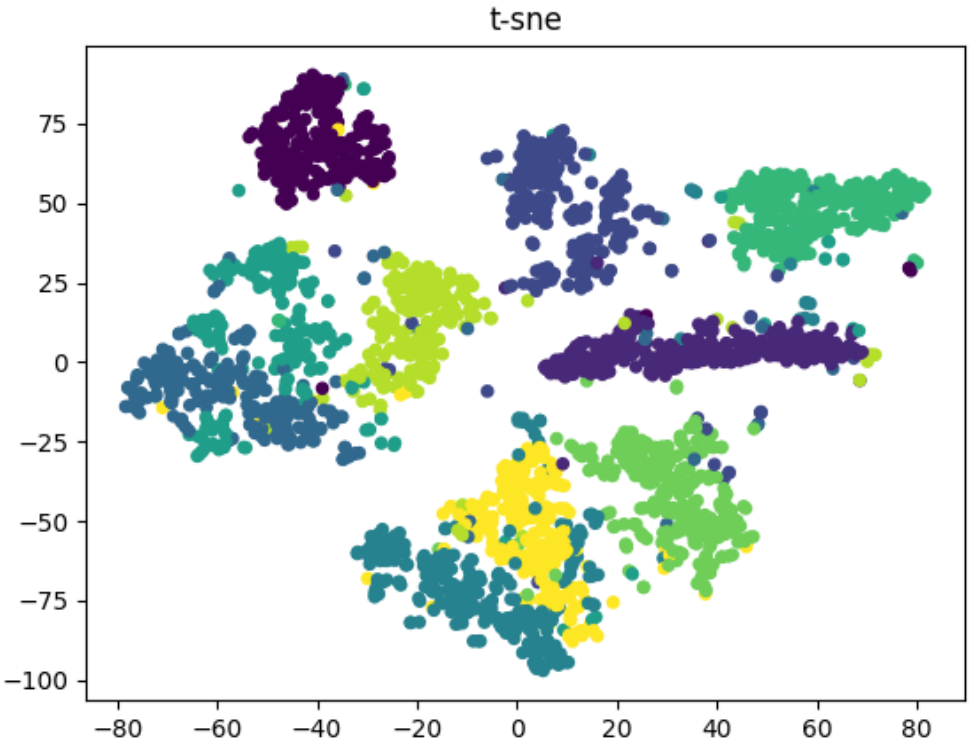
- Perplexity = 25 (imgur, imgur)



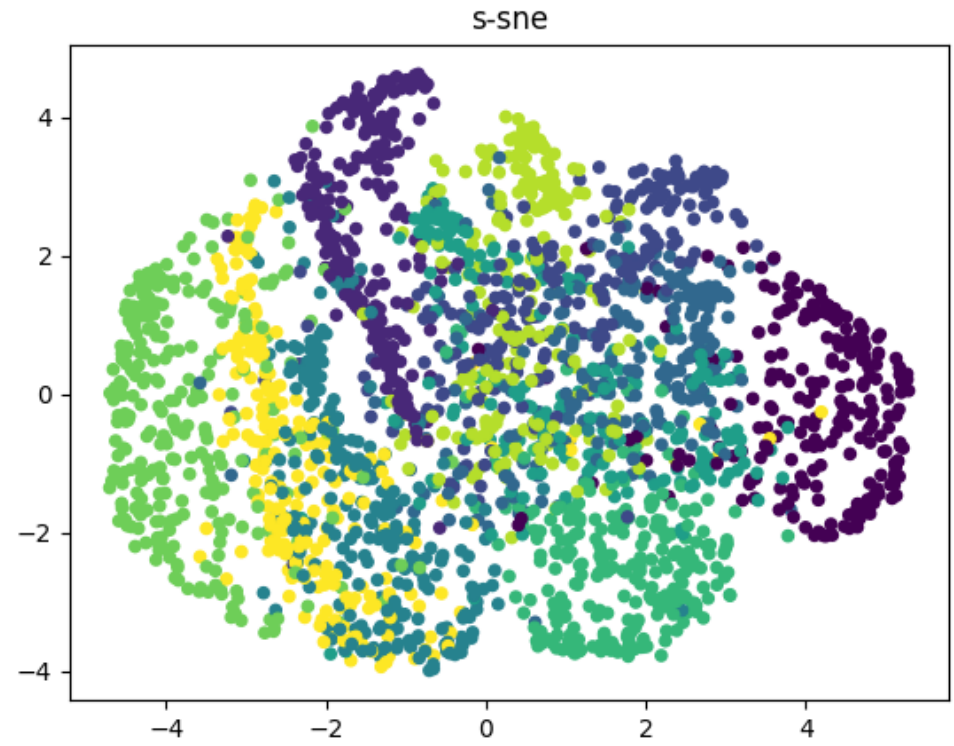


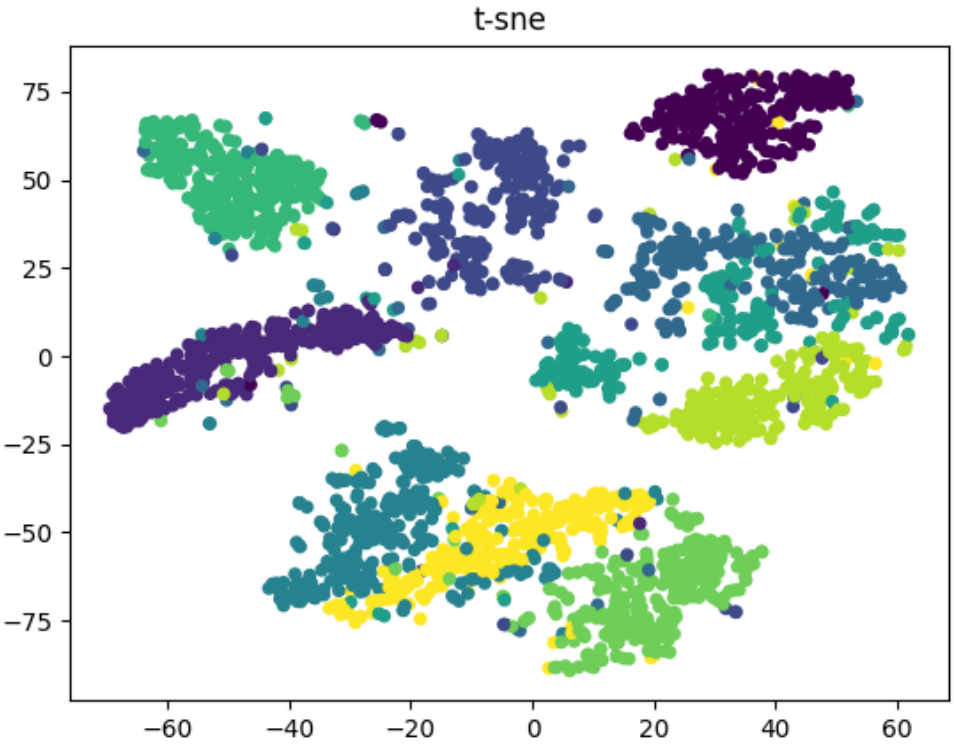
- Perplexity = 30 (imgur, imgur)





- Perplexity = 35 (imgur, imgur)





Observations and discussion

Meaning of eigenface

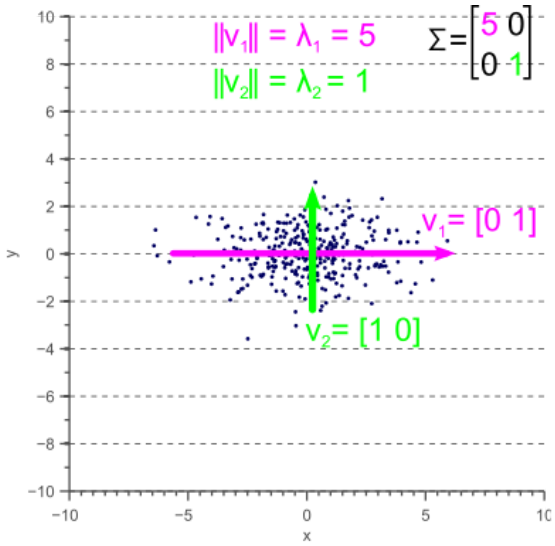
The eigenfaces (eigenvectors) is extracted from eigen decomposition of the covariance matrix of data X.

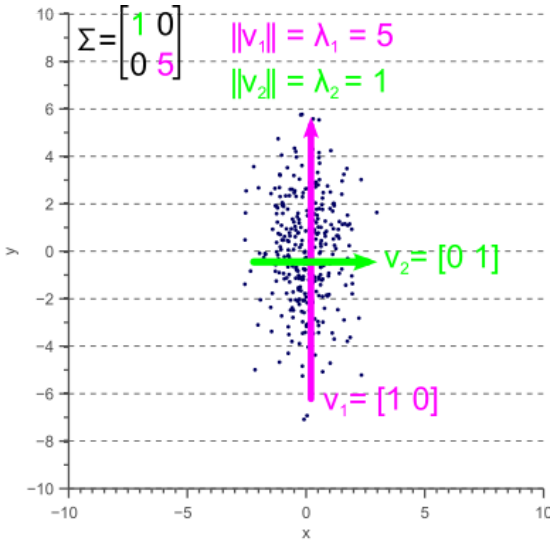
In the geometry meaning, this process is to extract the independent vectors (eigenvectors).

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

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

- Eigenvectors (eigenfaces)





- Eigenvalues versus covariance