Report

Kernel Eigenfaces

- Code with Detailed Explanation

  - Part 1: Simple PCA & LDA (eigenfaces and reconstruction)

    Procedures

    1. Load Data
    2. Do PCA, show eigenfaces, show reconstructions
    3. Do LDA, show eigenfaces, show reconstructions

```
# Load data
X_train, y_train = loadData(train_database)
X_test, y_test   = loadData(test_database)

# PCA
X_proj, PCs, X_mean = PCA(X_train, 25)
show_eigenfaces(PCs, 25)
show_reconstructions(X_train, np.matmul(X_proj, PCs.T), 10)
# LDA
X_proj, W = LDA(X_train, y_train, 25)
show_eigenfaces(W, 25)
show_reconstructions(X_train, np.matmul(X_proj, W.T), 10)
```

2 types of reconstructions based on PCA/LDA: the one below is shifted by mean.

```
# PCA
X_proj, PCs, X_mean = PCA(X_train, 25)
show_eigenfaces(PCs, 25)
show_reconstructions(X_train, np.matmul(X_proj, PCs.T)+X_mean, 10)
# LDA
X_proj, W = LDA(X_train-X_train.mean(axis=0), y_train, 25)
show_eigenfaces(W, 25)
show_reconstructions(X_train, np.matmul(X_proj, W.T)+X_train.mean(axis=0), 10)
```

Load data

Resize original images from (231, 195) to (77, 65), and flatten the images.

```
H, W = 77, 65
def loadData(_dir:str):
    image_files = os.listdir(_dir)
    N = len(image_files)
    data = np.zeros((N, H*W))
    labels = np.zeros((N,), dtype=np.uint8)

    for i, file in enumerate(image_files):
        img = Image.open(os.path.join(_dir, file)).resize((W,H), Image.ANTIALIAS)
        data[i,:] = np.asarray(img.getdata()).flatten() # H*W
        labels[i] = int(file.split('.')[0][7:9])-1

    return data, labels
```

Show eigenfaces

```
def show_eigenfaces(X, n_images):
    n = int(np.sqrt(n_images))
    fig, axes = plt.subplots(n, n, figsize=(8, 8))

    for i in range(n_images):
        ax = axes[int(i/n),int(i%n)]
        ax.imshow(X[:,i].reshape(H,W), cmap='gray')
        ax.axis('off')

    fig.patch.set_visible(False)
    fig.tight_layout()
    plt.show()
```

Show reconstructions

```
def show_reconstructions(X, X_recover, n_faces):
    fig, axes = plt.subplots(2, n_faces, figsize=(10, 4))

    idx = np.random.choice(X_recover.shape[0], n_faces)
    for i, face_idx in enumerate(idx):
        ax1 = axes[0,int(i%n_faces)]
        ax2 = axes[1,int(i%n_faces)]
        ax1.imshow(X[face_idx,:].reshape(H,W), cmap='gray')
        ax2.imshow(X_recover[face_idx,:].reshape(H,W), cmap='gray')
        ax1.axis('off')
        ax2.axis('off')

    fig.patch.set_visible(False)
    fig.tight_layout()
    plt.show()
```

PCA

Find an orthogonal projection $\mathbf{U}$ (principle components) in which the data $\mathbf{x}$ after projection $\mathbf{z} = \mathbf{x}\mathbf{U}$ will have maximum variance (i.e. min MSE).

1. Compute the sample mean and translate the data, so that it's centered around origin.
2. Compute the covariance matrix $\mathbf{S}$
3. Do eigendecomposition
   By maximizing the variance of projection, we have $\mathbf{S}\mathbf{u}_k = \lambda_1 \mathbf{u}_k$, so that we could do eigendecomposition on $\mathbf{S}$ to find principal components.
4. Pick the M first largest eigenvectors of $\mathbf{S}$ as **PCs**
   $\mathbf{U}$ is composed of the M first largest eigenvectors of covariance matrix $\mathbf{S}$ of $\mathbf{x}$.
   Note: $\mathbf{U}$ is **PCs** in the code below.
5. Project the centered data on to the space spanned by PCs
   $\mathbf{z} = \mathbf{x}\mathbf{U}$ here the dimension
6. Reconstruction
   $\hat{\mathbf{x}} = \mathbf{x}\mathbf{U}\mathbf{U}^\top$

```
def PCA(X, M):
    # Compute the sample mean and translate the data,
    # so that it's centered around origin.
    X_centered = X - X.mean(axis=0) # (N,P)
    # Compute the covariance matrix
    S = np.cov(X_centered.T) # (P,P)
    # Eigendecomposition
    eigenvalues, eigenvectors = LA.eigh(S)
    # Principal components are first m largest eigenvectors
    PCs = np.flip(eigenvectors[:, -M:], axis=1) # (P,M)
    # Project the centered data on to the space spanned by PCs
    X_proj = np.matmul(X_centered, PCs) # (N,P)

    return X_proj, PCs, X.mean(axis=0)
```

LDA

Project data onto a line (orthogonally), such that the class separation is maximized(, and variance of each class is minimized).

1. In order to accelerate computation (using the benefits of vectorization of numpy), use a matrix $\mathbf{R}$ to indicate each data belongs to which label
2. Calculate within-class scatter $\mathbf{S_W}$

$$\mathbf{S}_W = \sum_{k=1}^{K} \mathbf{S}_k = \sum_{k=1}^{K} \sum_{n \in C_k} (\mathbf{x}_n - \mathbf{m}_k)(\mathbf{x}_n - \mathbf{m}_k)^\top$$

where $\mathbf{m}_k = \frac{1}{N_k} \sum_{n \in C_k} \mathbf{x}_n$

3. Calculate between-class scatter $\mathbf{S_B}$

$$\mathbf{S}_B = \sum_{k=1}^{K} N_k (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^\top$$

4. To maximize $\mathbf{S_B}$ and minimize $\mathbf{S_W}$, solve $\mathbf{S}_W^{-1}\mathbf{S}_B\mathbf{w}_k = \lambda \mathbf{w}_k$ to get optimized weight matrix $\mathbf{W}$ which is formed by D' largest eigenvalues.
   Note that we should do psuedoinverse.
5. Project the data on to the space spanned by $\mathbf{W}$
   $\mathbf{z} = \mathbf{x}\mathbf{W}$ here the dimension
6. Reconstruction
   $\hat{\mathbf{x}} = \mathbf{x}\mathbf{W}\mathbf{W}^\top$

```
def LDA(X, y, D_):

    N, P = X.shape # N: n_data, P: n_dim
    K = len(np.unique(y)) # K: n_class

    # To indicate each data belongs to which label
    R = np.zeros((N,K)) # (N,K)
    R[np.arange(N), y] = 1
    Nk = R.sum(axis=0) # (K,)
    # Calsulate mean of each class and total class
    mk = np.matmul(R.T, X) / Nk[:,np.newaxis] # (K,P)
    m  = X.mean(axis=0) # shape=(P,)

    # Calculate SW
    xn_mk = X - np.matmul(R, mk) # (N,P)
    SW = np.matmul(xn_mk.T, xn_mk) # (P,P)

    # Calculate SB
    mk_m = mk - m # (K,P)
    SB = np.matmul(Nk * mk_m.T, mk_m) # (P,P)

    # Optimizw w by eigendecomposition of SW^{-1}SB
    S = np.matmul(LA.pinv(SW), SB)
    eigenvalues, eigenvectors = LA.eigh(S)

    W = np.flip(eigenvectors[:, -D_:], axis=1) # (P,D_)

    X_proj = np.matmul(X, W) # (N,D_)

    return X_proj, W
```

- Part 2: Simple PCA & LDA (face recognition, performance comparison)

PCA

1. Transform the test data (normalizing the mean) and project the translated test data onto dimensionality-reduced space by the principal components **PCs**.
   Note that since the mean of the training data have been translated to origin, we also have to translate the test data by minus the mean of training data
2. Do prediction and compute accuracy

```
# PCA
X_train_proj, PCs, X_train_mean = PCA(X_train, 25)
X_test_proj = np.matmul(X_test - X_train_mean, PCs)
prediction = KNN(X_train_proj, y_train, X_test_proj, k=5)
acc = performance(y_test, prediction)
print("ACC of PCA = {:.2f}%".format(acc*100))
```

LDA

1. Project the test data onto dimensionality-reduced space by **W**
2. Do prediction and compute accuracy

```
# LDA
X_train_proj, W = LDA(X_train, y_train, 25)
X_test_proj = np.matmul(X_test, W)
prediction = KNN(X_train_proj, y_train, X_test_proj, k=5)
acc = performance(y_test, prediction)
print("ACC of LDA = {:.2f}%".format(acc*100))
```

KNN

```
// Tr: training dataset, Ts: testing dataset, labels: Tr's labels, k: # nearest neighbors
Input: Tr, Ts, labels, k
Output: prediction
Begin
    For xn in Ts
        Compute the distances b/w the test data xn and Tr
        Pick k training data which are closest to xn and derive their corresponding labels.
        Do maximum voting to determine the final prediction
End
```

```
def KNN(X_train, y_train, X_test, k):
    max_vote = lambda x: np.argmax(np.bincount(x))

    prediction = np.zeros((X_test.shape[0],))
    for i, xn in enumerate(X_test):
        # Compute the distances b/w a test data xn and training dataset
        D = LA.norm(X_train-xn, axis=1)
        # Pick the k training data which are closest to xn,
        # and derive their corresponding labels
        k_candidates = y_train[np.argsort(D)[:k]]
        # do maximum voting to determine the final prediction
        prediction[i] = max_vote(k_candidates)

    return prediction
```

Perfomance

Compute the accuaracy

```
def performance(y_test, prediction):
    return np.sum(y_test == prediction)/y_test.shape[0]
```

- Part 3: Kernel PCA & LDA

Kernels

1. Linear kernel

   $K(x, x') = x^T x' + c$, where c is a coefficient.

2. RBF kernel

   $K(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$, where $\|x - x'\|^2$ is squared Euclidean distance between two feature vectors. σ is a free parameter. We can also rewrite it as following.

   Let $\gamma = \frac{1}{2\sigma^2} \Rightarrow K(x, x') = \exp(-\gamma\|x - x'\|^2)$

3. Linear kernel + RBF kernel

```
def linearKernel(X, X_):
    return np.matmul(X, X_.T)

def rbfKernel(X, X_, gamma=1e-4):
    return np.exp(-gamma*cdist(X, X_,'euclidean'))

def linear_plus_rbfKernel(X, X_, gamma=1e-4):
    return np.matmul(X, X_.T)+np.exp(-gamma*cdist(X, X_,'euclidean'))
```

Kernel PCA

First, assume $\Phi(X)$ are centered

$$\frac{1}{N}\Phi(\mathbf{X})\Phi(\mathbf{X})^\top\mathbf{u} = \lambda(u) \text{ with } \mathbf{u} = \sum_i \mathbf{a}_i\Phi(\mathbf{x}_i) = \Phi(\mathbf{X})\mathbf{a}$$

Plug in $\mathbf{u}$

$$\Phi(\mathbf{X})\Phi(\mathbf{X})^\top\Phi(\mathbf{X})\mathbf{a} = \lambda N\Phi(\mathbf{X})\mathbf{a}$$

Multiply with $\Phi(\mathbf{X})^\top$

$$\Phi(\mathbf{X})^\top\Phi(\mathbf{X})\Phi(\mathbf{X})^\top\Phi(\mathbf{X})\mathbf{a} = \lambda N\Phi(\mathbf{X})^\top\Phi(\mathbf{X})\mathbf{a}$$
$$\mathbf{K}\mathbf{K}\mathbf{a} = \lambda N\mathbf{K}\mathbf{a}$$
$$\mathbf{K}\mathbf{a} = \lambda N\mathbf{a}$$

Find principle components $\mathbf{a}_i$ by solving $\mathbf{K}\mathbf{a} = \lambda N\mathbf{a}$
Second, extend to general case

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

where $\mathbf{1}_N$ is NxN matrix with every element 1/N

Thus, solve $\mathbf{K}^{\text{Centered}}\mathbf{a} = \lambda N\mathbf{a}$
Project test kernel into principal components of kernel PCA

$$\Phi(\mathbf{x}_{new})\mathbf{U} = \Phi(\mathbf{X}_i)\Phi(\mathbf{x}_{new})\mathbf{a} = \mathbf{K}(\mathbf{X}, \mathbf{x}_{new})\mathbf{a}$$

Note that we should also translate test kernel by mean of train kernel.

```
def kernelPCA(X, M, kernel):
    K = kernel(X, X)
    N = K.shape[0]
    oneN = np.ones((N,N))/N
    K_centered = K - np.matmul(oneN, K) - np.matmul(K, oneN) + \
                 np.matmul(oneN, np.matmul(K, oneN))
    eigenvalues, eigenvectors = LA.eigh(K_centered/N)
    PCs = np.flip(eigenvectors[:, -M:], axis=1)
    K_proj = np.matmul(K_centered, PCs)

    return K_proj, PCs, K.mean(axis=0)

# Test kernel
K_test = linear_plus_rbfKernel(X_test, X_train)
# PCA
K_train_proj, PCs, K_train_mean = kernelPCA(X_train, 25, linear_plus_rbfKernel)
# Project test kernel into principal components of kernel PCA
K_test_proj = np.matmul(K_test-K_train_mean, PCs)

prediction = KNN(K_train_proj, y_train, K_test_proj, k=5)
acc = performance(y_test, prediction)
print("ACC of kernel PCA = {:.2f}%".format(acc*100))
```

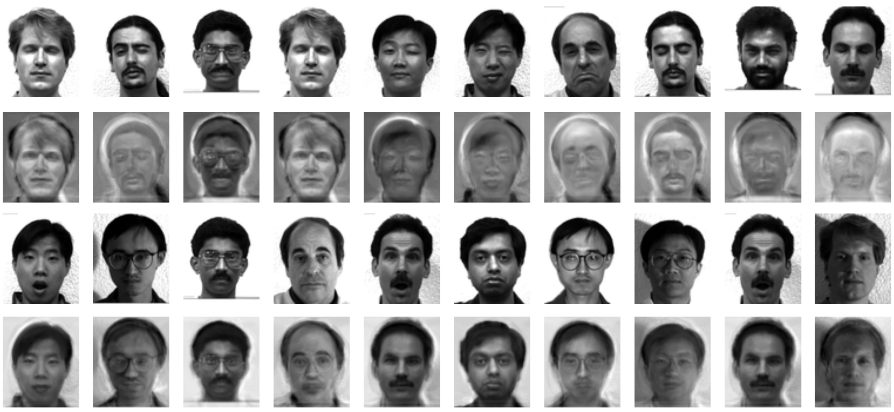
Kernel LDA

The only difference is the kernel trick, replace original X in Part 1. with precomputed kernel. We solve $\mathbf{S}_W^{\phi-1}\mathbf{S}_B^{\phi}\mathbf{w}_k = \lambda\mathbf{w}_k$ instead. Then we project test kernel by $\mathbf{W}$

```
def kernelLDA(X, y, D_, kernel):
    N, P = X.shape # N: n_data, P: n_dim
    K = len(np.unique(y)) # K: n_class

    Kernel = kernel(X, X)

    # To indicate each data belongs to which label
    R = np.zeros((N,K)) # (N,K)
    R[np.arange(N), y] = 1
    Nk = R.sum(axis=0) # (K,)

    # Calsulate mean of each class and total class
    mk = np.matmul(R.T, Kernel) / Nk[:,np.newaxis] # (K,P)
    m  = Kernel.mean(axis=0) # shape=(P,)

    # Calculate SW
    xn_mk = Kernel - np.matmul(R, mk) # (N,P)
    SW = np.matmul(xn_mk.T, xn_mk) # (P,P)

    # Calculate SB
    mk_m = mk - m # (K,P)
    SB = np.matmul(Nk * mk_m.T, mk_m) # (P,P)
```

```
    # Optimizw w by eigendecomposition of SW^{-1}SB
    S = np.matmul(LA.pinv(SW),SB)
    eigenvalues, eigenvectors = LA.eigh(S)

    W = np.flip(eigenvectors[:, -D_:], axis=1) # (P,D_)

    Kernel_proj = np.matmul(Kernel, W) # (N,D_)

    return Kernel_proj, W

# Test kernel
K_test = linear_plus_rbfKernel(X_test, X_train)
# LDA
K_train_proj, W = kernelLDA(X_train, y_train, 25, linear_plus_rbfKernel)
# Project test kernel by W
K_test_proj = np.matmul(K_test, W)
prediction = KNN(K_train_proj, y_train, K_test_proj, k=5)
acc = performance(y_test, prediction)
print("ACC of kernel LDA = {:.2f}%".format(acc*100))
```

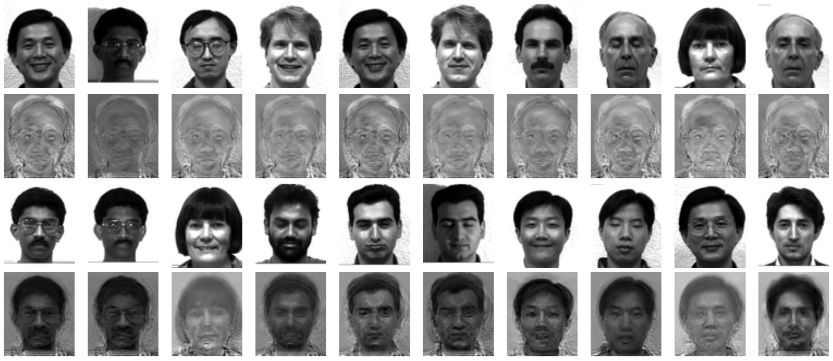
- Experiment settings and results & discussions & observations
  - Part 1: Simple PCA & LDA (eigenfaces and reconstruction)
    PCA: 25 Eigenfaces and randomly choose 10 reconstructions



Recovered X_centered

Recovered X_centered + X_mean
shift back to original position

LDA: 25 fisherfaces and randomly choose 10 reconstructions



Recovered X

Recovered X_centered + X_mean
shift back to original position

We can see that the eigenfaces from PCA present clear different part of features of human's face, yet fisherfaces from LDA present less unclear (from my observation, the differences among fisherfaces are little). I found some papers that improve fisherfaces, such as dot product PCA and LDA eigenvectors as new LDA eigenvectors. Each eigenvector explains a part of variance and the eigenvector with larger eigenvalue can explain more variance, thereby, we can see that the corresponding eigenfaces provide different significant features of human's face.

Like how we have a first glance on or memorize people's face. If asked to distinguish different faces, we must recognize the contour of their faces or skin colors first.

Secondly, let's compare their results of reconstruction. It's clear and reasonable that the reconstructions of PCA are rather better than reconstruction of LDA from our previous observation between eigenfaces and fisherfaces. The recovered X of LDA are all approximately look like the elderly except the second and fourth men, yet we still can find that it reconstructs some important features for the second and fourth men. Nonetheless, compared with reconstructions of LDA, that of PCA sufficiently contains the significant features that explaining how a person will look like and be distinguished from others. In addition, the reconstruction loss of PCA is significantly lower than that of LDA.

In addition, for LDA, I also compute the total scatter and found that $\mathbf{S_T = S_w + S_B}$ holds. Therefore, we can either compute $(\mathbf{S_W}, \mathbf{S_B})$, $(\mathbf{S_T}, \mathbf{S_W})$, or $(\mathbf{S_T}, \mathbf{S_B})$

- Part 2: Simple PCA & LDA (face recognition, performance comparison)

KNN	ACC of PCA	ACC of LDA
K=1	83.33%	83.33%
K=3	83.33%	93.33%
K=5	90.00%	93.33%
K=7	90.00%	96.67%
K=9	80.00%	93.33%

Although from Part1 we see that PCA looks better than LDA, the accuracies of LDA are better than PCA under different K settings. Additionally, when K=7, their accuracies are the best, yet when K continuously increases, their ACCs drop. I speculate that since the number of classes is 15 and each class has 9 data, thereby the better K may be less than 9. To avoid being vulnerable to few neighbors (avoid high variance), we also cannot choose 1 nearest data as our prediction

- Part 3: Kernel PCA & LDA
From observation in Part 2, let's use K=7 for KNN.

Kernel	Gamma	ACC of PCA	ACC of LDA
None		90.00%	96.67%
Linear		83.33%	56.67%
RBF	1e−6	86.67%	76.67%
	1e−5	86.67%	76.67%
	1e−4	83.33%	80.00%
	1e−3	43.33%	40.00%
	1e−2	16.67%	10.00%
	1e−1	6.67%	13.33%
	1e+0	6.67%	16.67%
	1e+1	6.67%	16.67%
Linear+RBF	1e−6, 1e−5, 1e−4, 1e−3, 1e−2, 1e−1, 1e+0, 1e+1	80.00%	60.00%

I found that Kernel PCA & LDA are much faster than PCA & LDA.
For the ACC comparison b/w simple and kernel PCA & LDA, we can see that both of the simple one is better than kernel versions (only on my three types of kernels).

Besides, by tuning the parameter γ in RBF kernel, from observations, the ACCs of LDA are like a convex function with minimum ACC in the middle γ=1e−2.

In addition, although simple PCA performs slightly worse than simple LDA, kernel PCA performs much better than LDA (under linear kernel, linear_RBF kernel, and RBF kernel when $\gamma \leq 1e{-}2$)

I didn't tuning the weights for linear+RBF kernel, which resulted in that linear part dominates the results; yet RBF part still plays an important role to modulate the results, which we can observe the different results between linear and linear+RBF kernels.

For overall observation, PCA perfoms better than LDA most of the time; and from the

t-SNE

- Code with Detailed Explanation
  - Part 1:

    Understanding t-SNE

    t-SNE

```
// X: H-dim data, no_dim: # dim in L-dim, init_dim: # dim to simplify X,
// Perp: perplexity (cost function parameters), Y: L-dim data
Input : X, no_dim, init_dim, Per
Output: Y
Begin
    Do PCA to reduce original dim to init_dim
    Compute H-dim pairwise distances p_ij with Perp __eq(1)
    Randomly sample init solution Y, and set max_iter = 1000
    For t = 1 to max_iter do
    Begin
        Compute L-dim pairwise affinities q_ij __eq(2)
        Compute gradient __eq(3)
        Update __eq(4)
    End
End
Equation 1:
```

$$p_{ij} = p_{ji} = \frac{p_{j|i} + p_{i|j}}{2N} \quad \text{where} \quad p_{j|i} = \frac{\exp\left(- \left\| x_j - x_i \right\|^2 / 2\sigma_i^2\right)}{\sum_{k \neq l} \exp\left(- \left\| x_k - x_i \right\|^2 / 2\sigma_i^2\right)}$$

```
Equation 2:
```

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

```
Equation 3:
```

$$\nabla y = \frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i}^{N} \left(p_{ji} - q_{ji}\right)\left(y_i - y_j\right)\left(1 + \left\| y_i - y_j \right\|^2\right)^{-1}$$

Equation 4:

$$y^{(t+1)} = y^{(t)} + \Delta^{(t)}$$
$$\text{where } \Delta^{t} = -\rho^{(t)}\eta\nabla y^{(t)} + \alpha\Delta^{(t-1)}$$

α: momentum that take advantage of what we learned before.
ρ: gain ratio to monitor and tune the step size (learning rate) η

$$\rho^{(t)} = \begin{cases} \rho^{(t-1)} \times 0.8, \text{if } \nabla y^{(t)}, \Delta y^{(t-1)} \text{ is same direction} \\ \rho^{(t-1)} + 0.2 \text{ if } \nabla y^{(t)}, \Delta y^{(t-1)} \text{ is opposite direction} \\ \rho^{(0)} = 1.0 \end{cases}$$

$$\alpha = \begin{cases} 0.5, t \leq 250 \\ 0.8, t > 250 \end{cases}, \quad \alpha \geq 0.01$$

```
def tsne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0):
    """
        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 isinstance(no_dims, float):
        print("Error: array X should have type float.")
        return -1
    if round(no_dims) != no_dims:
        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))

    # Compute P-values
    P = x2p(X, 1e-5, perplexity)
    P = P + np.transpose(P)
    P = P / np.sum(P)
    P = P * 4.                                    # early exaggeration
    P = np.maximum(P, 1e-12)
```

```
# Run iterations
for iter in range(max_iter):

    # Compute pairwise affinities
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
    num[range(n), range(n)] = 0.
    Q = num / np.sum(num)
    Q = np.maximum(Q, 1e-12)

    # Compute gradient
    PQ = P - Q
    for i in range(n):
        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.) != (iY > 0.)) + \
            (gains * 0.8) * ((dY > 0.) == (iY > 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:
        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.

# Return solution
return Y
```

Given perplexity, tune and find proper σ_i for similarity/probability $p_{j|i}$ of each data.

```
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 each data point, and use binary search to find σ_i

```
    # 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.
                else:
                    beta[i] = (beta[i] + betamax) / 2.
            else:
                betamax = beta[i].copy()
                if betamin == np.inf or betamin == -np.inf:
                    beta[i] = beta[i] / 2.
                else:
                    beta[i] = (beta[i] + betamin) / 2.

            # 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
```

Hbeta: compute perplexity for the β_i given by binary search

$$Perp\left(P_i\right) = e^{H(P_i)}$$

$$H\left(P_i\right) = -\sum_{j=1}^{N} p_{j|i} \ln p_{j|i} = \ln \sum_{k \neq i} \exp\left(-\beta_i \left\|x_k - x_i\right\|^2\right) + \frac{\sum_{j=1}^{N} \exp\left(-\beta_i \left\|x_j - x_i\right\|^2\right) \cdot \beta_i \left\|x_j - x_i\right\|^2}{\sum_{k \neq i} \exp\left(-\beta_i d_{ki}^2\right)}$$

where $\beta_i = \dfrac{1}{2\sigma_i^2}$

```
def Hbeta(D=np.array([]), beta=1.0):
    """
        Compute the perplexity and the P-row for a specific value of the
        precision of a Gaussian distribution.
    """

    # Compute P-row and corresponding perplexity
    P = np.exp(-D.copy() * beta)
    sumP = sum(P)
    H = np.log(sumP) + beta * np.sum(D * P) / sumP
    P = P / sumP
    return H, P
```

Symmetric SNE

The differences b/w t-SNE and s-SNE are as below

Equation 2': Affinities Q follows Gaussian distribution,
yet Q for t-SNE follows student t-distribution.

$$q_{ji} = q_{ij} = \frac{\exp\left(- \| y_j - y_i \|^2\right)}{\sum_{k \neq l} \exp\left(- \| y_k - y_l \|^2\right)} \quad \text{(s-SNE)}$$

$$q_{ji} = q_{ij} = \frac{\left(1 + \| y_j - y_i \|^2\right)^{-1}}{\sum_{k \neq l} \left(1 + \| y_k - y_l \|^2\right)^{-1}} \quad \text{(t-SNE)}$$

```
# Compute pairwise affinities Q
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y))
num[range(n), range(n)] = 0.
Q = num / np.sum(num)
Q = np.maximum(Q, 1e-12)
```

Equation 3': Gradient for update formula. Since their cost functions are different due to the difference b/w their probability distribution in L-dim , their derived gradients are different as below.

$$\nabla y = \frac{\partial C}{\partial y_i} = 4 \sum_{j=1}^{N} \left(p_{ij} - q_{ij}\right)\left(y_i - y_j\right) \quad \text{(s-SNE)}$$

$$\nabla y = \frac{\partial C}{\partial y_i} = 4 \sum_{j \neq i}^{N} \left(p_{ji} - q_{ji}\right)\left(y_i - y_j\right)\left(1 + \| y_i - y_j \|^2\right)^{-1} \quad \text{(t-SNE)}$$

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

For symmetric SNE, p_{ij} will be small if x_i is an outlier such that $\|x_i - x_j\|^2$ is large, which leads to little contribution of y_i to cost C.

We want all data points contribute equally.

In addition, the local structure (neighbor relationships) of s-SNE is constrained by the degree of freedom in L-dim, which means that Gaussian is not appropriate due to the lack of consideration of degree of freedom.

When the output dimensionality < the effective dimensionality of input data (internal DoF), the neighborhoods are mismatched, which causes crowding problem in L-dim.

That is, in H-dim, we can have more points equidistant from each other, which may cause problems in L-dim.

1-dim: at most 2 points/2-dim: at most 3 points/n-dim: at most n+1 points can be equidistant

- Part 2
  Using scatterplot to visualize embedding for both s-SNE and t-SNE, and color each point by corresponding label. (Visualizing every iterations)

```
def visualizeEmbedding(Y, labels, method, perplexity, iter):
    fig = plt.figure(figsize=(4,4))
    ax = fig.add_subplot(111)
    ax.set_title("{} with Perp={}".format(method, int(perplexity)))
    ax.scatter(Y[:, 0], Y[:, 1], 20, labels)
    ax.axis('off')
    fig.savefig("./media/embedding/{}_{}_{}.png".format(method, int(perplexity), iter),
                format="png", dpi=300, bbox_inches="tight")
    plt.close(fig)
```

- Part 3
  Using heatmap to plot P (probability matrix in H-dim) and Q's (probability matrix in L-dim) distribution. Note that I rescale them by logarithm. Besides, I sorted P and Q by labels when plotting heatmaps for easier observation.

```
def visualizeAffinities(P, Q, labels, method, perplexity):
    fig, ax = plt.subplots(1, 2, figsize=(10,4))

    idx = labels.argsort()
    P_sorted = P[:,idx][idx]
    Q_sorted = Q[:,idx][idx]

    ax[0].set_title('P (H-dim)')
    ax[0].imshow(np.log(P_sorted), cmap='gray')
    ax[0].axis('off')
```

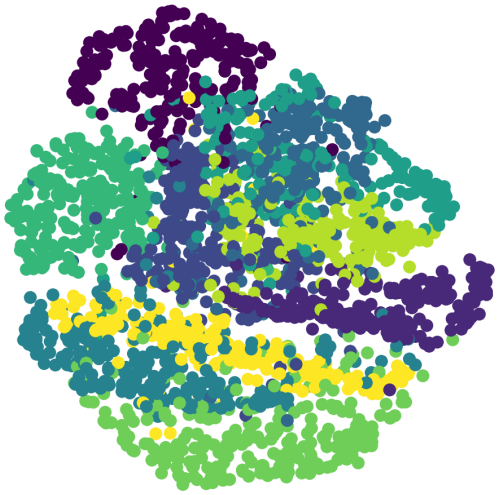
```
    ax[1].set_title('Q (H-dim)')
    ax[1].imshow(np.log(Q_sorted), cmap='gray')
    ax[1].axis('off')

    fig.patch.set_visible(False)
    fig.tight_layout()
    fig.savefig("./media/heatmap_{}_{}.png".format(method, int(perplexity)))
```

- Part 4
  I hand-tuned perplexity as the last parameter below, and pass it to s-SNE, t-SNE, and visualizations.

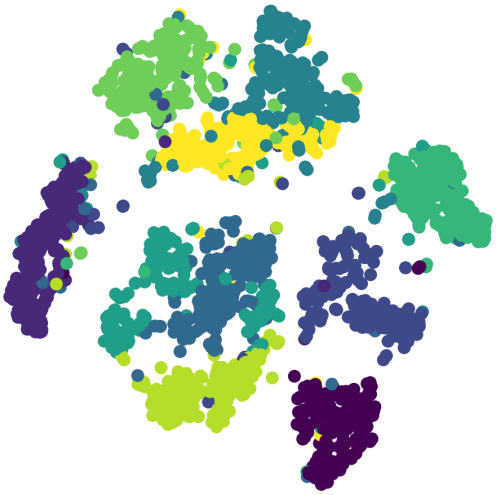
```
Y = ssne(X, 2, 50, 20.0)
visualizeEmbedding(Y, labels, "s-SNE", perplexity, iter)
visualizeAffinities(P, Q,labels, "s-SNE", perplexity)
```

- Experiment settings and results & discussions & observations
  - Part 1
    Set max_iter=1000, perplexity=20

s-SNE with Perp=20



t-SNE with Perp=20



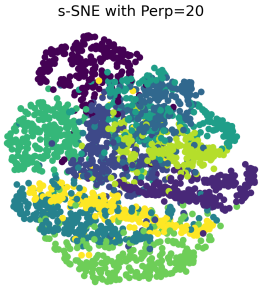
From the results of s-SNE and t-SNE, the crowding problem in s-SNE did happen, and t-SNE did soothe the crowding problem nicely by using student t-distribution as probability distribution in L-dim.

As mentioned in the code explanation, for s-SNE, the neighbor relationships are constrained in L-dim with the lack of consideration of DoF, and the corresponding y_i of an outlier x_i contributes quite little to the cost C. Consequently, the contributions of all points are unbalanced/unequal, such that the crowding problem happened.
If considering how well they map the data and preserve the relationships in H-dim, t-SNE performs better than s-SNE, and it did alleviate the crowding problem.

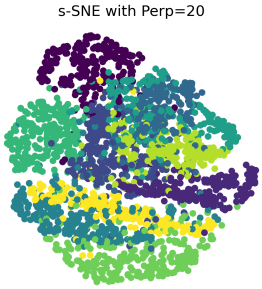
- Part 2

  Set max_iter=1000, perplexity=20
  From the videos I generated in ./AS7-2/videos/

  For s-SNE, it approximately didn't change its results after 175th iterations.

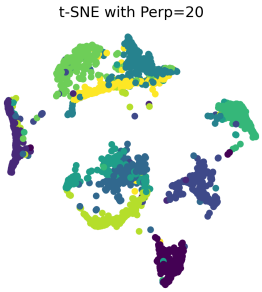


iter=175

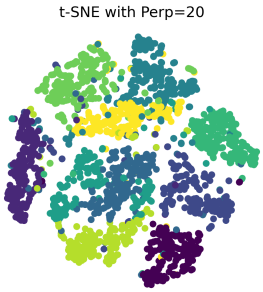


iter=1000

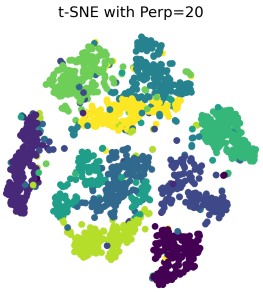
  For t-SNE, it first converged to a local optimum at 240th iterations, and jumped out of that local minimum and still have slightly changes until reaching the max iteration.



iter=100



iter=240



iter=500

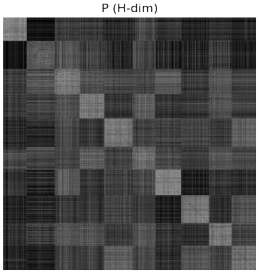
iter=1000

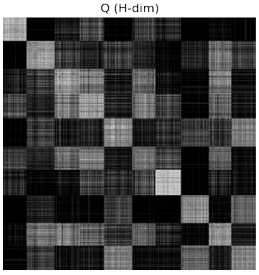
  To sum up, both of them reduced dimensions greatly that enclose similar data points and separate dissimilar data points. However, s-SNE converged earlier than t-SNE, yet it had crowding problem—s-SNE cannot further dissimilar data points much.

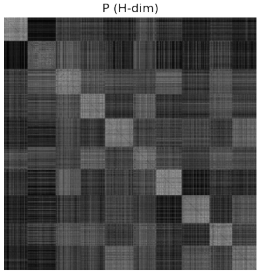
- Part 3

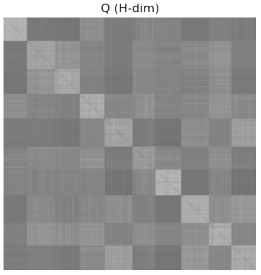
  Set max_iter=1000, perplexity=20
  The first two in the left are distributions of pairwise similarities P (H-dim) and Q (L-dim) based on s-SNE, and the last two in the right are that based on t-SNE.

  In overall, both of s-SNE and t-SNE preserve the relative similarities from P (H-dim). In other words, they approximately keep the local structure from P in Q.





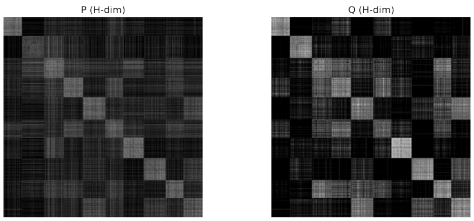
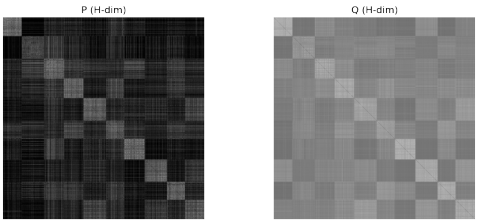
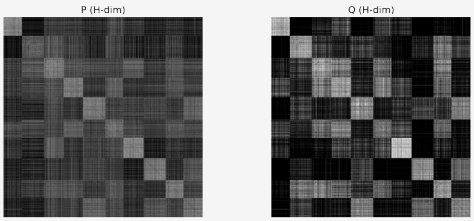
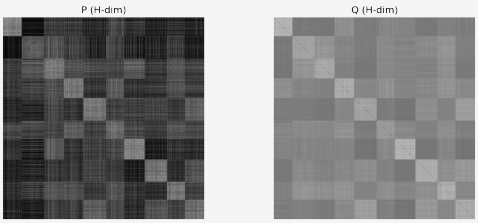
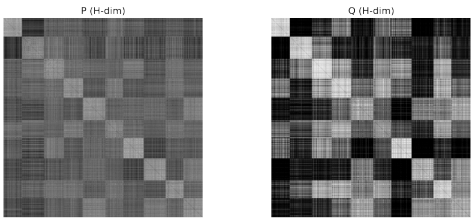
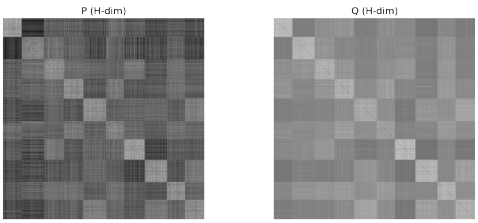
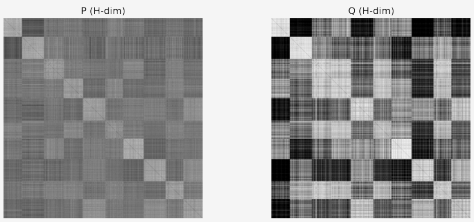
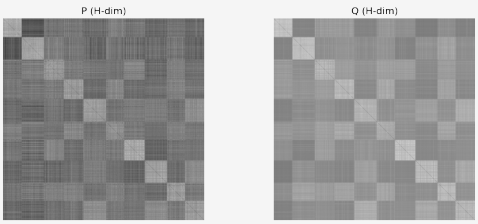
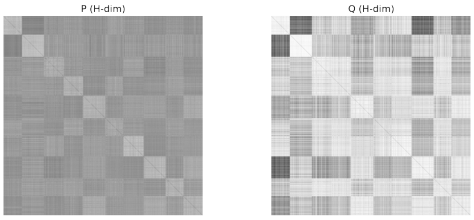
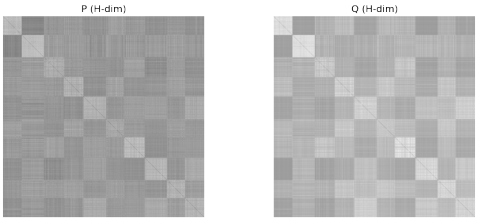
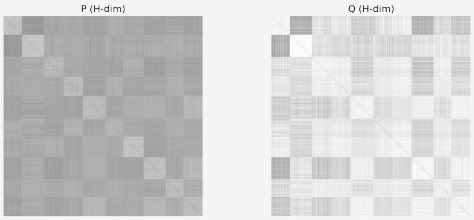
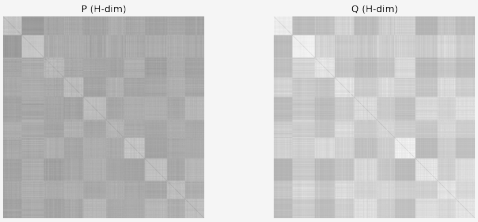




However, heatmap of Q in s-SNE is darker and more contrast than that in t-SNE. (i.e. the values in Q of s-SNE are in overall lower and more variant than that of t-SNE) In addition, t-SNE keeps neighbor relationships better than s-SNE since the distributions of P and Q are more alike than that of s-SNE.

Colormap: dark is low, bright is high.

- Part 4
  Set different perplexities in [10, 20, 50, 100, 500, 1000]

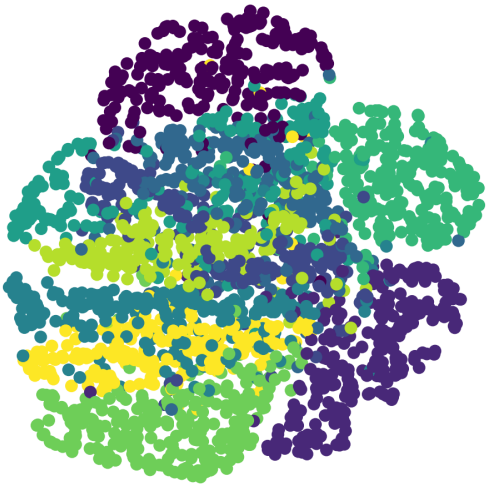
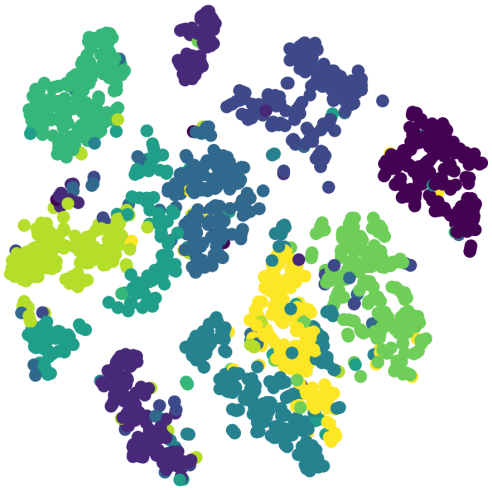
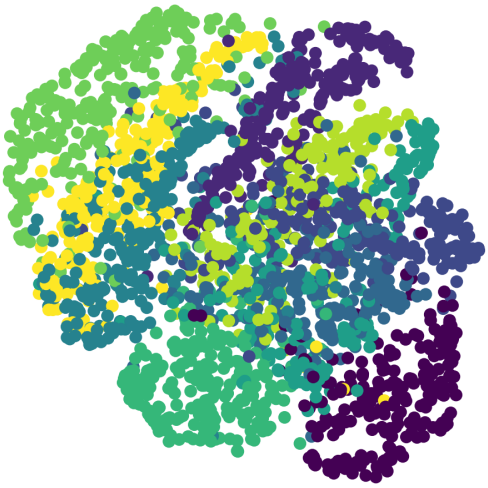
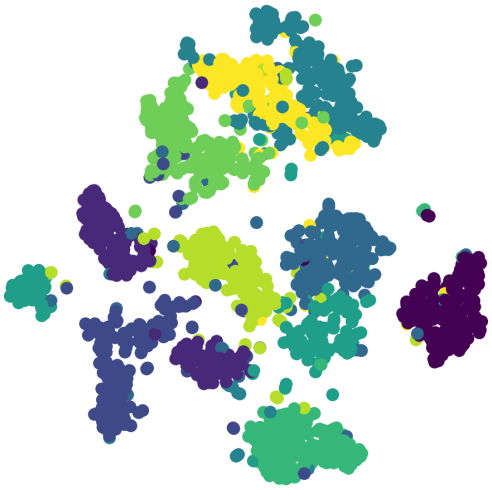
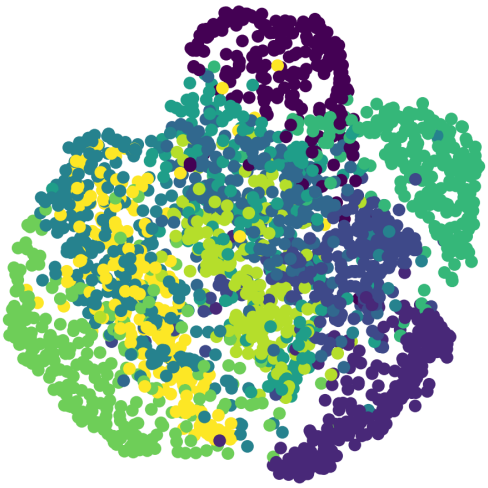
Perp	s-SNE	t-SNE
10		
20		
50		
100		
500		
1000		

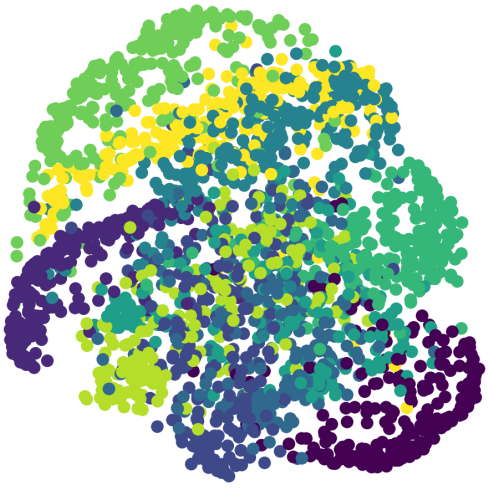
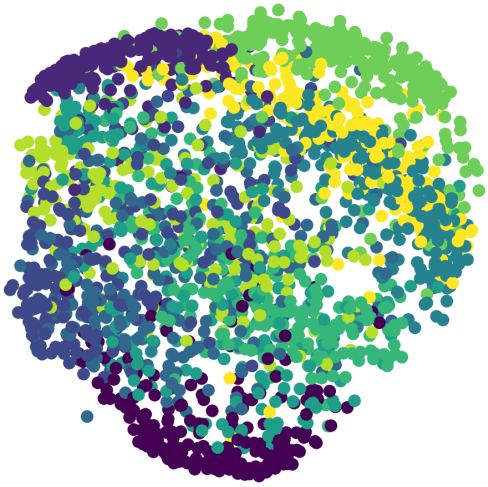
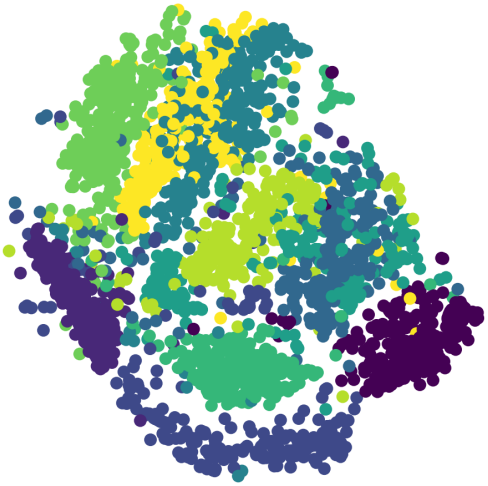
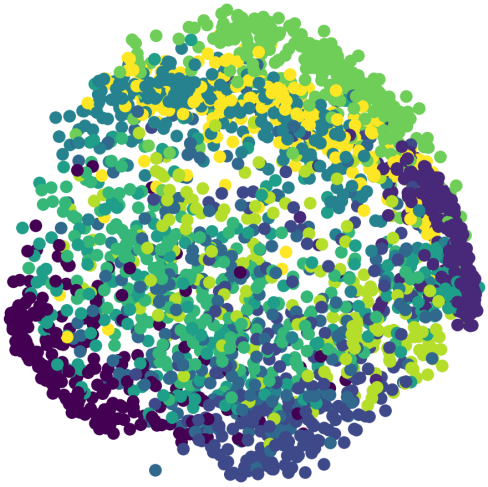
Visualization of distribution as above:
The values in both of P and Q based on s-SNE and t-SNE become larger(brighter) as perplexity increases.

For s-SNE, its Q also becomes more contrast and less similar to P from perplexity =10 to 100. However, t-SNE well preserves the neighbor relationships, and seems more robust to perplexity than s-SNE. (i.e. s-SNE is vulnerable to the change of perplexity) In addition, in my opinion, t-SNE preserves best at perplexity = 100. (P and Q are more alike)

Visualization of embeddings as below:
As the perplexity increases, there are more overlapped points in low dimension for both s-SNE and t-SNE. It's clear that since perplexity is a measure that describe the effective number neighbors. Thus, like K nearest neighbor, we may want to choose a proper number of neighbors. If perplexity in SNE becomes too large, we may choose several ineffective neighbors and make the boundaries unclear.

Compared s-SNE with t-SNE, s-SNE does encounter crowding problem no matter what the perplexity is, but the crowding problem becomes more severe as the perplexity increase; t-SNE seems to encounter crowding problem when the perplexity becomes too large. As a result, perplexity from 10 to 100 will be proper for t-SNE under this dataset.

Perp	s-SNE	t-SNE
10	s-SNE with Perp=10 	t-SNE with Perp=10 
20	s-SNE with Perp=20 	t-SNE with Perp=20 
50	s-SNE with Perp=50 	t-SNE with Perp=50 

Perp	s-SNE	t-SNE
100	s-SNE with Perp=100 	t-SNE with Perp=100 
500	s-SNE with Perp=500 	t-SNE with Perp=500 
1000	s-SNE with Perp=1000 	t-SNE with Perp=1000