ML HW 7 Report

Original Report link

May not be showed properly in pdf. https://hackmd.io/tYvpTytfRs2onJCyTzH6-w

Code with detailed explanations

Kernel eigenfaces

Part 1

```
def PCA(X, dims):
    mu = np.mean(X, axis=0)
    print(mu.shape)
    cov = (X - mu).T @ (X - mu)
    eigen_val, eigen_vec = np.linalg.eigh(cov)
    print(eigen_vec.shape)
    #eigen_vec = (X - mu).T     @ eigen_vec
    print(eigen_vec.shape)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :dims].real
    print(W)
    return [W, mu]
```

PCA的目的是保留data最大的variance，因此，我們必須先構建出data的covariance matrix，根據Rayleigh quotient，保留最大variance的projection方向就是covariance matrix最大的eigenvalue對應的eigenvector。因此，依題目要求我們要降成25維，就是取covariance matrix的前25個eigenvector並歸一化。

```
def LDA(X, label, dims):
    (n, d) = X.shape
    label = np.asarray(label)
    c = np.unique(label)
    mu = np.mean(X, axis=0)
    S_w = np.zeros((d, d), dtype=np.float64)
    S_b = np.zeros((d, d), dtype=np.float64)
    for i in c:
        X_i = X[np.where(label == i)[0], :]
        mu_i = np.mean(X_i, axis=0)
        S_w += (X_i - mu_i).T @ (X_i - mu_i)
        S_b += X_i.shape[0] * ((mu_i - mu).T @ (mu_i - mu))
    eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(S_w) @ S_b)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    print(eigen_vec.shape)
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :dims].real
    return W

```

LDA的目的是盡可能放大Between class variance，並進可能縮小Within class variance，因此我們必須先構建SB和SW。SB和SW構建方法如下。一樣by Rayleigh quotient，取Sw**-1@SB的前25個eigenvector並歸一化。

if now we are dealing with multi-class cases $(\mathcal{C}_1, \mathcal{C}_2, \cdots, \mathcal{C}_k)$:

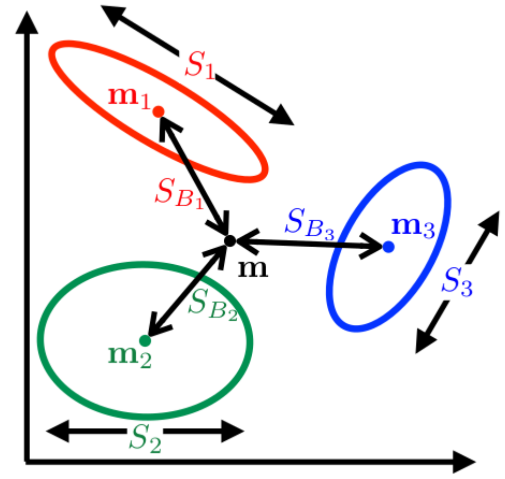
$$\textit{within-class scatter}:\ S_W = \sum_{j=1}^{k} S_j,\ \text{where } S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$$

$$\text{and } \mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} x_i$$

between-class scatter:

$$S_B = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

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



```
def draw(target_data, target_filename, title, W, mu=None):
    if mu is None:
        mu = np.zeros(target_data.shape[1])
    projection = (target_data - mu) @ W
    reconstruction = projection @ W.T + mu
    folder = f"{title}_{datetime.now().strftime('%Y%m%d-%H%M%S')}"
    os.mkdir(folder)
    os.mkdir(f'{folder}/{title}')
    if W.shape[1] == 25:
        plt.clf()
        for i in range(5):
            for j in range(5):
                idx = i * 5 + j
                plt.subplot(5, 5, idx + 1)
                plt.imshow(W[:, idx].reshape(SHAPE[::-1]), cmap='gray')
                plt.axis('off')
        plt.savefig(f'./{folder}/{title}/{title}.png')
    for i in range(W.shape[1]):
        plt.clf()
        plt.title(f'{title}_{i + 1}')
        plt.imshow(W[:, i].reshape(SHAPE[::-1]), cmap='gray')
        plt.savefig(f'./{folder}/{title}/{title}_{i + 1}.png')

    if reconstruction.shape[0] == 10:
        plt.clf()
        for i in range(2):
            for j in range(5):
                idx = i * 5 + j
                plt.subplot(2, 5, idx + 1)
                plt.imshow(reconstruction[idx].reshape(SHAPE[::-1]),
cmap='gray')
                plt.axis('off')
        plt.savefig(f'./{folder}/reconstruction.png')
    for i in range(reconstruction.shape[0]):
        plt.clf()
        plt.title(target_filename[i])
        plt.imshow(reconstruction[i].reshape(SHAPE[::-1]), cmap='gray')
        plt.savefig(f'./{folder}/{target_filename[i]}.png')
```

找出basis後，我們可以拿這個basis來proj和reconstruct，並輸出結果。reconstruct的方法為將降維後的結果和每個basis做linear combination並加上本來的mean得出。 $\hat{\mathbf{X}} = \mathbf{Z}\mathbf{V}^\top = \mathbf{X}\mathbf{V}\mathbf{V}^\top$.

Part 2

```
W, mu = PCA(data, 25)
X_proj = (X - mu) @ W
test_proj = (test - mu) @ W
faceRecognition(X_proj, X_label, test_proj, test_label, 'PCA')

W = LDA(data, label, 25)
X_proj = X @ W
test_proj = test @ W
faceRecognition(X_proj, X_label, test_proj, test_label, 'LDA')
```

先對data做PCA/LDA得到projection basis後，把train/test data減去平均並和basis內積得到projection vector，並送到facerecognition裡做KNN classification。

```
def faceRecognition(X, X_label, test, test_label, method,
kernel_type=None):
    if kernel_type is None:
        print(f'Face recognition with {method} and KNN:')
    else:
        print(f'Face recognition with Kernel {method}({kernels[kernel_type
- 1]}) and KNN:')
    dist_mat = []
    for i in range(test.shape[0]):
        dist = []
        for j in range(X.shape[0]):
            dist.append((distance(X[j], test[i]), X_label[j]))
        dist.sort(key=lambda x: x[0])
        dist_mat.append(dist)
    for k in K:
        correct = 0
        total = test.shape[0]
        for i in range(test.shape[0]):
            dist = dist_mat[i]
            neighbor = np.asarray([x[1] for x in dist[:k]])
            neighbor, count = np.unique(neighbor, return_counts=True)
            predict = neighbor[np.argmax(count)]
            if predict == test_label[i]:
                correct += 1
        print(f'K={k:>2}, accuracy: {correct / total:>.3f}
({correct}/{total})')
    print()
```

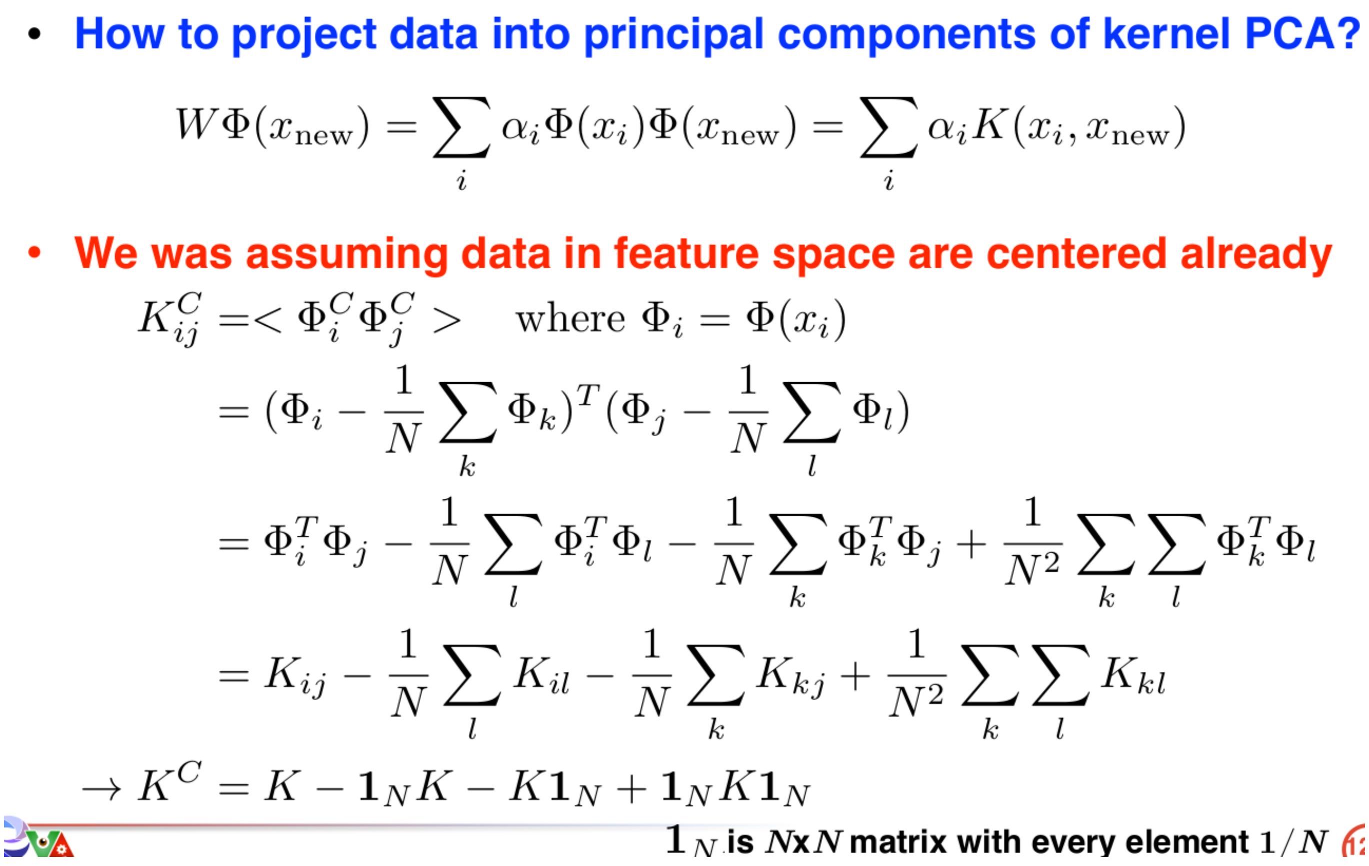
先計算所有test data和train data的相對距離，對每筆test data選出和它離最近的k個train data，讓這k個data的label以多數決predict此test data的label並計算accuracy。

Part 3

```
def kernelPCA(X, dims, kernel_type):
    kernel = getKernel(X, kernel_type)
    n = kernel.shape[0]
    one = np.ones((n, n), dtype=np.float64) / n
    kernel = kernel - one @ kernel - kernel @ one + one @ kernel @ one
    eigen_val, eigen_vec = np.linalg.eigh(kernel)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    eigen_val = np.column_stack(eigen_val[idx])
    W = eigen_vec[:, idx][:, :dims].real
    return kernel @ W
```

kernel PCA可用做非線性分離的case上。我們不確定feature space是不是centered的，我們必須先按下面公式建出feature space的covariance matrix K，並按PCA的方式取K的前25個eigenvector並歸一化作為basis。

project後的coordinate就是kernel @ W。



- **How to project data into principal components of kernel PCA?**

$$W\Phi(x_{\text{new}}) = \sum_i \alpha_i \Phi(x_i)\Phi(x_{\text{new}}) = \sum_i \alpha_i K(x_i, x_{\text{new}})$$

- **We was assuming data in feature space are centered already**

$$\begin{aligned} K_{ij}^C &= < \Phi_i^C \Phi_j^C > \quad \text{where } \Phi_i = \Phi(x_i) \\ &= (\Phi_i - \frac{1}{N}\sum_k \Phi_k)^T(\Phi_j - \frac{1}{N}\sum_l \Phi_l) \\ &= \Phi_i^T\Phi_j - \frac{1}{N}\sum_l \Phi_i^T\Phi_l - \frac{1}{N}\sum_k \Phi_k^T\Phi_j + \frac{1}{N^2}\sum_k\sum_l \Phi_k^T\Phi_l \\ &= K_{ij} - \frac{1}{N}\sum_l K_{il} - \frac{1}{N}\sum_k K_{kj} + \frac{1}{N^2}\sum_k\sum_l K_{kl} \end{aligned}$$

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

$\mathbf{1}_N$ **is** N**x**N **matrix with every element** $1/N$

```
def kernelLDA(X, label, dims, kernel_type):
    label = np.asarray(label)
    c = np.unique(label)
    kernel = getKernel(X, kernel_type)
    n = kernel.shape[0]
    mu = np.mean(kernel, axis=0)
    N = np.zeros((n, n), dtype=np.float64)
    M = np.zeros((n, n), dtype=np.float64)
    for i in c:
        K_i = kernel[np.where(label == i)[0], :]
        l = K_i.shape[0]
        mu_i = np.mean(K_i, axis=0)
        N += K_i.T @ (np.eye(l) - (np.ones((l, l), dtype=np.float64) / l))
@ K_i
        M += l * ((mu_i - mu).T @ (mu_i - mu))
    eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(N) @ M)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :dims].real
    return kernel @ W
```

$$\mathbf{w}=\sum_{i=1}^{N}\alpha_i\phi\left(\mathbf{x}_i\right)$$

從上課的內容中可以知道w是data在feature space中的線性組合。

$$\mathbf{w}^{\mathrm{T}}\mathbf{m}_k^{\phi}=\frac{1}{N_k}\sum_{i=1}^{N}\sum_{j=1}^{N_k}\alpha_i k\left(\mathbf{x}_i,\mathbf{x}_j^k\right)=\alpha^{\mathrm{T}}\mathbf{M}_k$$

其中 $(\mathbf{M}_k)_j=\frac{1}{N_k}\sum_{j=1}^{N_k}k(\mathbf{x}_i,\mathbf{x}_j^k)$

那么，现在我们可以重写 $\mathbf{w}^{\mathrm{T}}\mathbf{S}_B^{\phi}\mathbf{w}$

$$\mathbf{w}^{\mathrm{T}}\mathbf{S}_B^{\phi}\mathbf{w}=\mathbf{w}^{\mathrm{T}}\left(\mathbf{m}_2^{\phi}-\mathbf{m}_1^{\phi}\right)\left(\mathbf{m}_2^{\phi}-\mathbf{m}_1^{\phi}\right)^{\mathrm{T}}\mathbf{w}=\alpha^{\mathrm{T}}\mathbf{M}\alpha$$

其中 $\mathbf{M}=(\mathbf{M}_2-\mathbf{M}_1)(\mathbf{M}_2-\mathbf{M}_1)$

回頭看看上面那張老師kernel PCA的投影片，使用kernel trick時，我們要用aplha來project，因此這裡將SB轉為M。

$$\begin{aligned}
\mathbf{w}^{\mathrm{T}}\mathbf{S}_W^{\phi}\mathbf{w}&=\left(\sum_{i=1}^{N}\alpha_i\phi^{\mathrm{T}}(\mathbf{x}_i)\right)\left(\sum_{k=1,2}\sum_{n=1}^{N_k}\left(\phi(\mathbf{x}_n^k)-\mathbf{m}_k^{\phi}\right)\left(\phi(\mathbf{x}_n^k)-\mathbf{m}_k^{\phi}\right)^{\mathrm{T}}\right)\left(\sum_{j=1}^{N}\alpha_j\phi(\mathbf{x}_j)\right)\\
&=\sum_{k=1,2}\sum_{i=1}^{N}\sum_{n=1}^{N_k}\sum_{k=1}^{N}\left(\alpha_i\phi^{\mathrm{T}}(\mathbf{x}_i)\left(\phi(\mathbf{x}_n^k)-\mathbf{m}_k^{\phi}\right)\left(\phi(\mathbf{x}_n^k)-\mathbf{m}_k^{\phi}\right)^{\mathrm{T}}\alpha_j\phi(\mathbf{x}_j)\right)\\
&=\sum_{k=1,2}\sum_{i=1}^{N}\sum_{n=1}^{N_k}\sum_{k=1}^{N}\left(\alpha_i k(\mathbf{x}_i,\mathbf{x}_n^k)-\frac{1}{N_k}\sum_{p=1}^{N_k}\alpha_i k(\mathbf{x}_i,\mathbf{x}_p^k)\right)\left(\alpha_j k(\mathbf{x}_j,\mathbf{x}_n^k)-\frac{1}{N_k}\sum_{q=1}^{N_k}\alpha_j k(\mathbf{x}_j,\mathbf{x}_q^k)\right)\\
&=\sum_{k=1,2}\left(\sum_{i=1}^{N}\sum_{n=1}^{N_k}\sum_{j=1}^{N}\left(\alpha_i\alpha_j k(\mathbf{x}_i,\mathbf{x}_n^k)k(\mathbf{x}_j,\mathbf{x}_n^k)-\frac{2\alpha_i\alpha_j}{N_k}\sum_{p=1}^{N_k}k(\mathbf{x}_i,\mathbf{x}_n^k)k(\mathbf{x}_j,\mathbf{x}_p^k)+\frac{\alpha_i\alpha_j}{N_k^2}\sum_{p=1}^{N_k}\sum_{q=1}^{N_k}k(\mathbf{x}_i,\mathbf{x}_p^k)k(\mathbf{x}_j,\mathbf{x}_q^k)\right)\right)\\
&=\sum_{k=1,2}\left(\sum_{i=1}^{N}\sum_{n=1}^{N_k}\sum_{j=1}^{N}\left(\alpha_i\alpha_j k(\mathbf{x}_i,\mathbf{x}_n^k)k(\mathbf{x}_j,\mathbf{x}_n^k)-\frac{\alpha_i\alpha_j}{N_k}\sum_{p=1}^{N_k}k(\mathbf{x}_i,\mathbf{x}_n^k)k(\mathbf{x}_j,\mathbf{x}_p^k)\right)\right)\\
&=\sum_{k=1,2}\alpha^{\mathrm{T}}\mathbf{K}_k\mathbf{K}_k^{\mathrm{T}}\alpha-\alpha^{\mathrm{T}}\mathbf{K}_k\mathbf{1}_{N_k}\mathbf{K}_k^{\mathrm{T}}\alpha\\
&=\alpha^{\mathrm{T}}\mathbf{N}\alpha.
\end{aligned}$$

其中 $\mathbf{N}=\sum_{k=1,2}\mathbf{K}_k(\mathbf{I}-\mathbf{1}_{N_k})\mathbf{K}_k^{\mathrm{T}}$ ，其中 $\mathbf{I}$ 是一个单位矩阵， $\mathbf{1}_{N_k}$ 是一个每一个元素都为 $1/N_k$ 的矩阵， $\mathbf{K}_k$ 满足 $\mathbf{K}_k(a,b):=k\left(\mathbf{x}_a,\mathbf{x}_b^{(k)}\right)$ 。

將SW轉為N後，接著就和一般的LDA相同，找N**-1 @ M的前25個eigenvector得到basis，並用內積project接著再套用face recognition輸出結果。 ref : https://zhuanlan.zhihu.com/p/92359921

t-sne

Part 1

```
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.
                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
```

用我們給的perplexity算出High-D的P。

```
def sne(X=np.array([]), no_dims=2, initial_dims=50, perplexity=30.0,
method='tsne'):
    """
        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))
    interval = 50
    # 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
        if method == 'tsne':
            num = 1 / (1 + scipy.spatial.distance.cdist(Y, Y,
'sqeuclidean'))
        else:
            num = np.exp(-1 * scipy.spatial.distance.cdist(Y, Y,
'sqeuclidean'))
        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))
        if iter % interval == 0:
            plotResult(Y, labels, iter, interval, method, perplexity)
        # 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, P, Q
```

若是t-sne，low-D的Q是依據student-t distribution建的，若為sne，則類似高斯分佈。我們使用1000個iters讓Q的分佈不斷靠近P，即使KL(P||Q)盡量變小。

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

上圖為sne

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

上圖為t-sne

Part 2

```
def plotResult(Y, labels, idx, interval, method, perplexity):
    plt.clf()
    scatter = plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.legend(*scatter.legend_elements(), loc='lower left', title='Digit')
    plt.title(f'{method}, perplexity: {perplexity}, iteration: {idx}')
    plt.tight_layout()
    if interval:
        plt.savefig(f'./{method}_{perplexity}/{idx // interval}.png')
    else:
        plt.savefig(f'./{method}_{perplexity}/{idx}.png')
```

我們每50個iters call一次plotResult，使用plt.scatter依據label將low-D分成不同顏色畫出來。

```
def make_gif(method, perplexity):
    img_path = []
    for i in range(20):
        img_path.append(f'./{method}_{perplexity}/{i}.png')
    img_path.append(f'./{method}_{perplexity}/final.png')
    gif_images = []
    for path in img_path:
        gif_images.append(imageio.imread(path))
    imageio.mimsave("result.gif",gif_images,fps=1)
```

最後，使用這個function將前面的21張圖做成gif。

Part 3

```
def plotHighDLowD(P, Q, method, perplexity):
    plt.clf()
    pylab.title('tSNE high-dim')
    pylab.hist(P.flatten(),bins=40,log=True)
    pylab.savefig(f'./{method}_{perplexity}/High-D.png')
    pylab.show()
    plt.clf()
    pylab.title('tSNE low-dim')
    pylab.hist(Q.flatten(),bins=40,log=True)
    pylab.savefig(f'./{method}_{perplexity}/Low-D.png')
    pylab.show()
```

我們用pylab.hist將P和Q的分佈畫出來

Part 4

```
print("Run Y = tsne.tsne(X, no_dims, perplexity) to perform t-SNE on your
dataset.")
    print("Running example on 2,500 MNIST digits...")
    X = np.loadtxt("mnist2500_X.txt")
    labels = np.loadtxt("mnist2500_labels.txt")
    method = 'sne'
    perplexity = 30.0
    if not os.path.exists(f'{method}_{perplexity}'):
            os.mkdir(f'{method}_{perplexity}')
    Y, P, Q = sne(X, 2, 50, perplexity=perplexity, method=method)
    plotResult(Y, labels, 'final', None, method, perplexity)
    print(P)
    print(Q)
    plotHighDLowD(P, Q, method, perplexity)
    make_gif(method, perplexity)
```

在main-function裡使method和perplexity可調。

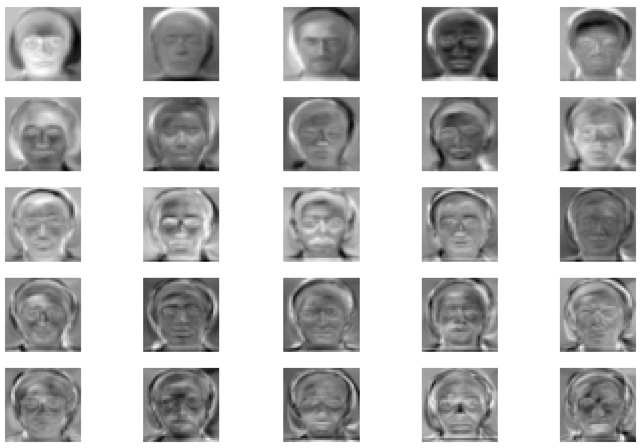
Experiments settings and results & discussion

Kernel eigenfaces

Part 1

PCA

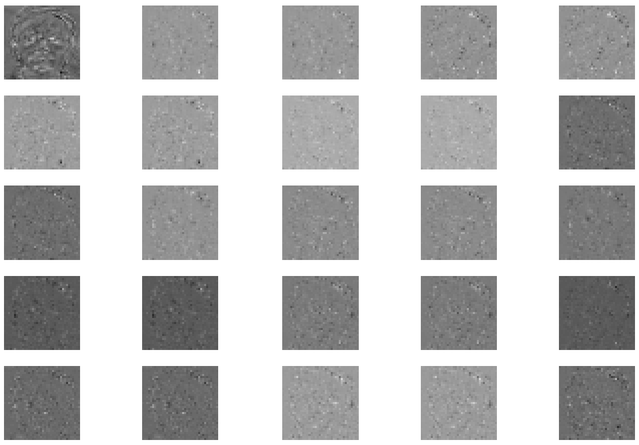
first 25 eigenfaces



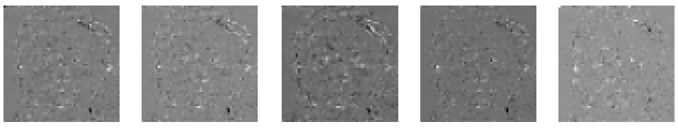
random 10 reconstruction

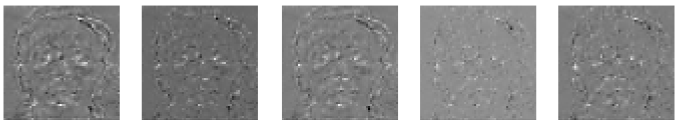
LDA

first 25 fisherfaces



random 10 reconstruction



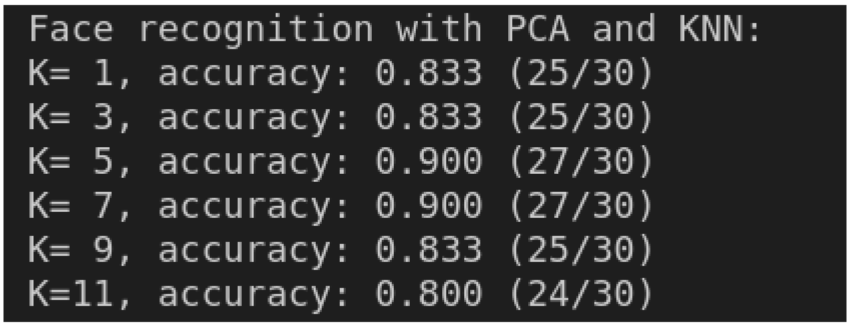


Discussion

我們可以看到，eigenfaces 效果比fisherfaces好。主要原因是 PCA 關注最大variance，這使我們能夠capture所有圖像之間的特徵，並保留每張圖不同的地方。而LDA考慮between-class和within-class的比率，它的目的是方便用於分類而不是重構(reconstruction)。

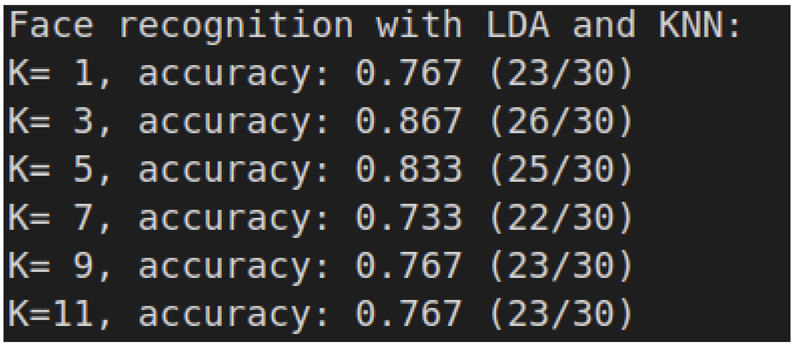
Part 2

PCA & KNN results



```
Face recognition with PCA and KNN:
K= 1, accuracy: 0.833 (25/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.900 (27/30)
K= 7, accuracy: 0.900 (27/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.800 (24/30)
```

LDA & KNN results



```
Face recognition with LDA and KNN:
K= 1, accuracy: 0.767 (23/30)
K= 3, accuracy: 0.867 (26/30)
K= 5, accuracy: 0.833 (25/30)
K= 7, accuracy: 0.733 (22/30)
K= 9, accuracy: 0.767 (23/30)
K=11, accuracy: 0.767 (23/30)
```

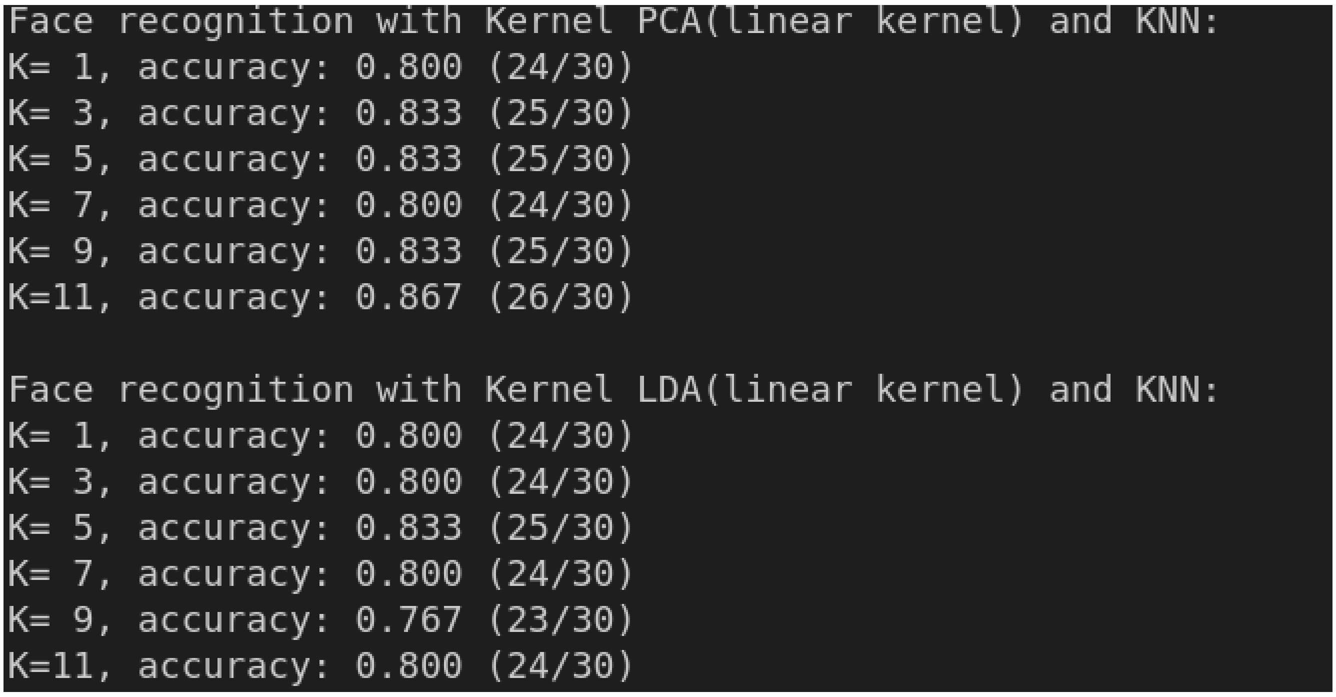
Discussion

看起來PCA的效果比較好，這跟預先猜想的不太一樣，畢竟LDA多了一項label的資訊，我同時也上網查了一些資料，大部分也都顯示LDA的準確度應該比較高，但也有一些資料顯示當資料過少時依據label投影有可能會有

一些overfitting的情況出現。

Part 3

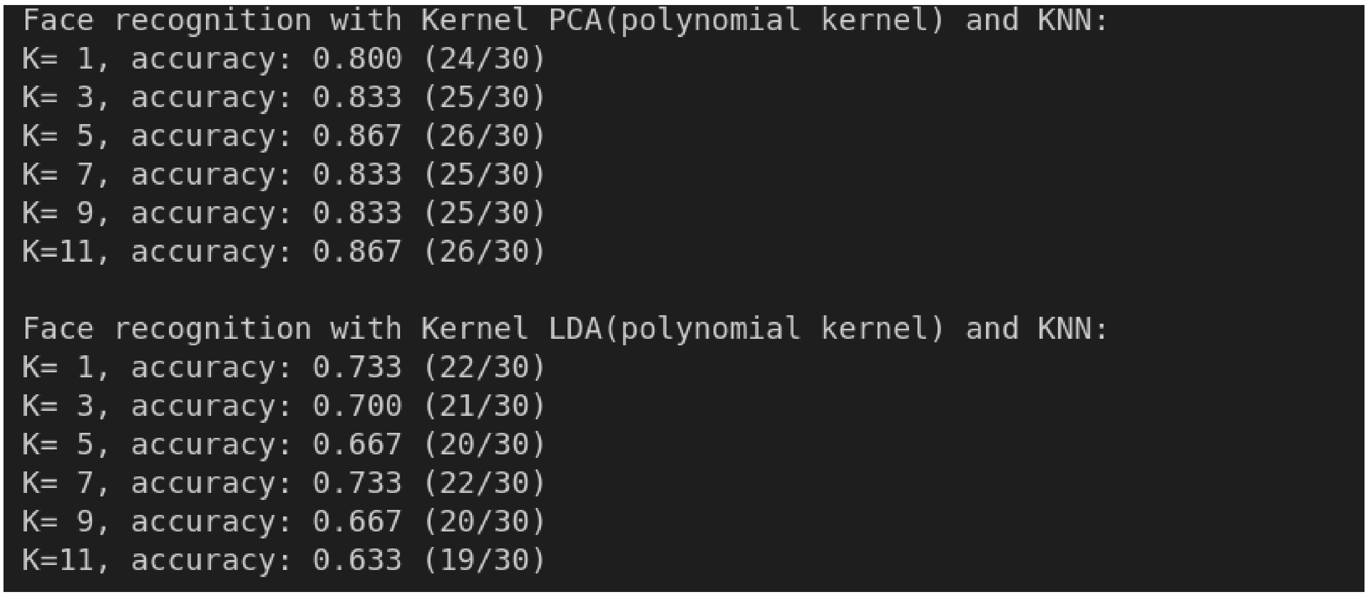
Linear kernel



```
Face recognition with Kernel PCA(linear kernel) and KNN:
K= 1, accuracy: 0.800 (24/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.833 (25/30)
K= 7, accuracy: 0.800 (24/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.867 (26/30)

Face recognition with Kernel LDA(linear kernel) and KNN:
K= 1, accuracy: 0.800 (24/30)
K= 3, accuracy: 0.800 (24/30)
K= 5, accuracy: 0.833 (25/30)
K= 7, accuracy: 0.800 (24/30)
K= 9, accuracy: 0.767 (23/30)
K=11, accuracy: 0.800 (24/30)
```

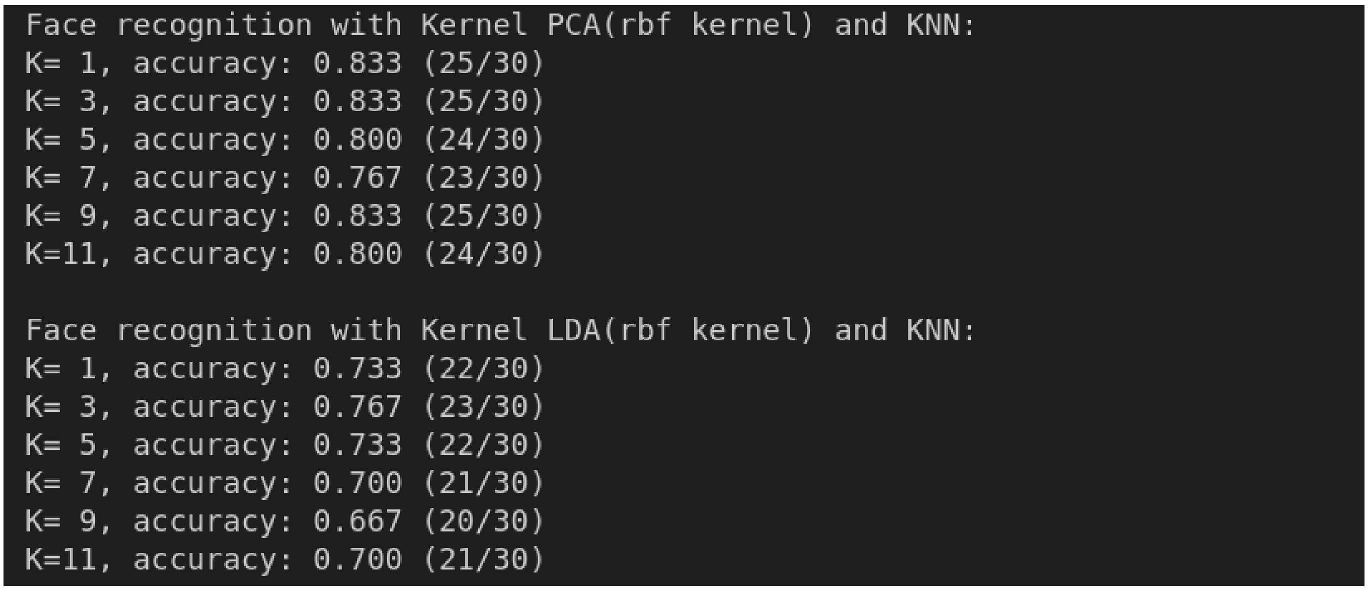
Polynomial kernel



```
Face recognition with Kernel PCA(polynomial kernel) and KNN:
K= 1, accuracy: 0.800 (24/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.867 (26/30)
K= 7, accuracy: 0.833 (25/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.867 (26/30)

Face recognition with Kernel LDA(polynomial kernel) and KNN:
K= 1, accuracy: 0.733 (22/30)
K= 3, accuracy: 0.700 (21/30)
K= 5, accuracy: 0.667 (20/30)
K= 7, accuracy: 0.733 (22/30)
K= 9, accuracy: 0.667 (20/30)
K=11, accuracy: 0.633 (19/30)
```

rbf kernel



```
Face recognition with Kernel PCA(rbf kernel) and KNN:
K= 1, accuracy: 0.833 (25/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.800 (24/30)
K= 7, accuracy: 0.767 (23/30)
K= 9, accuracy: 0.833 (25/30)
K=11, accuracy: 0.800 (24/30)

Face recognition with Kernel LDA(rbf kernel) and KNN:
K= 1, accuracy: 0.733 (22/30)
K= 3, accuracy: 0.767 (23/30)
K= 5, accuracy: 0.733 (22/30)
K= 7, accuracy: 0.700 (21/30)
K= 9, accuracy: 0.667 (20/30)
K=11, accuracy: 0.700 (21/30)
```

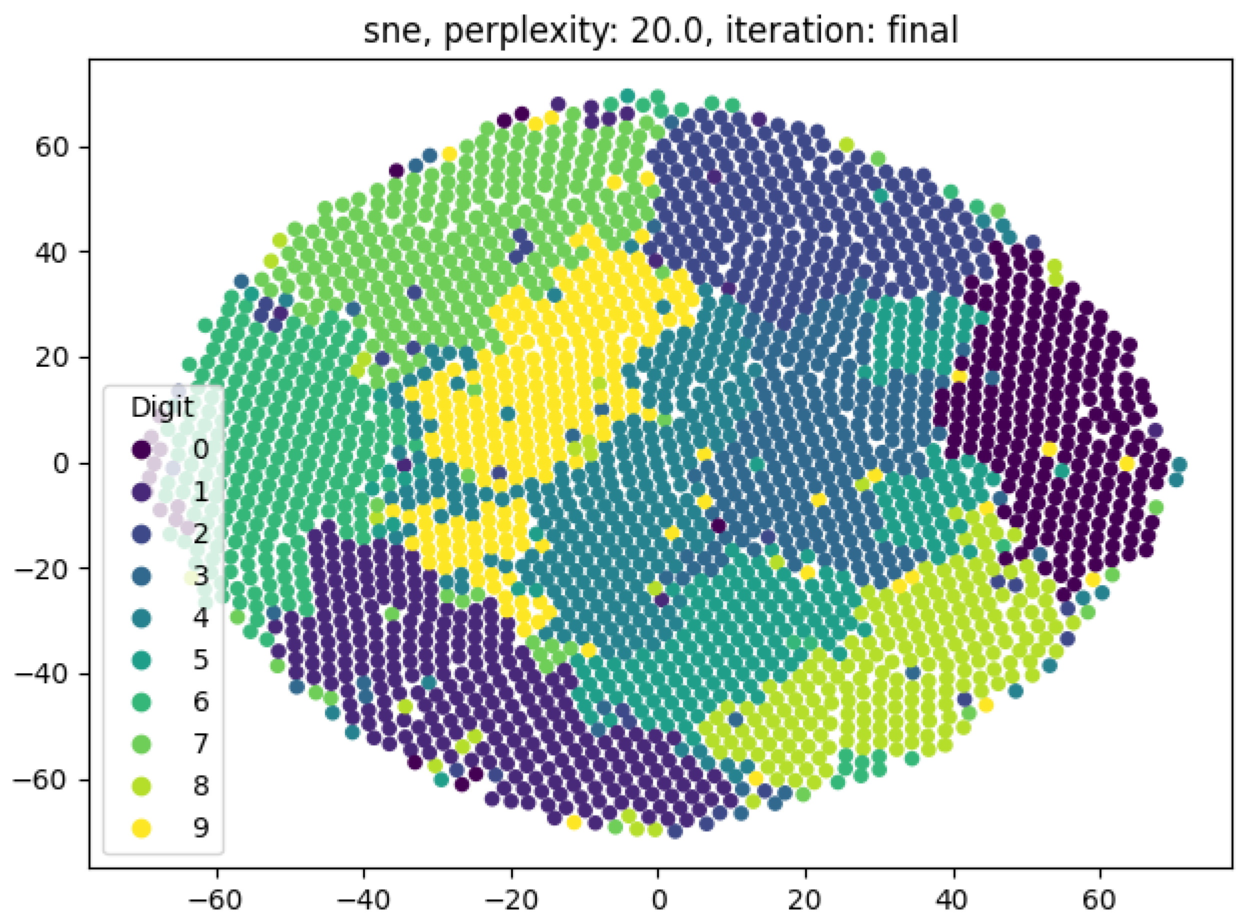
Discussion

Kernel PCA/LDA的表現都沒有原本的PCA好，這想起來並不合理，想來想去才發現這些kernel的hyperparameter對accuracy會有很大的影響，若多花點時間調參，準確度應該可以提昇，之後有空做實驗再更新md在github上。

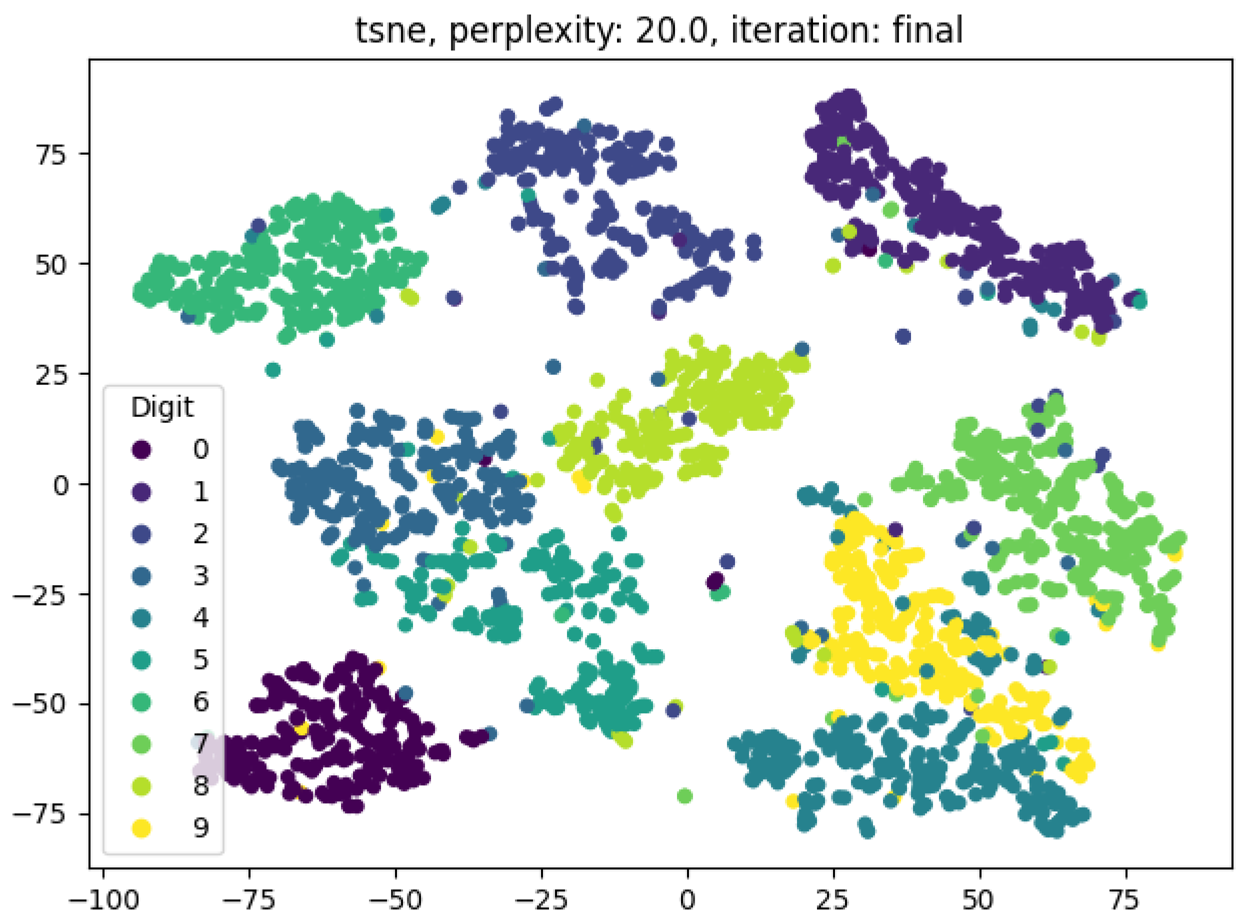
t-sne

Part 1

sne result(perplexity:20)



t-sne result(perplexity:20)

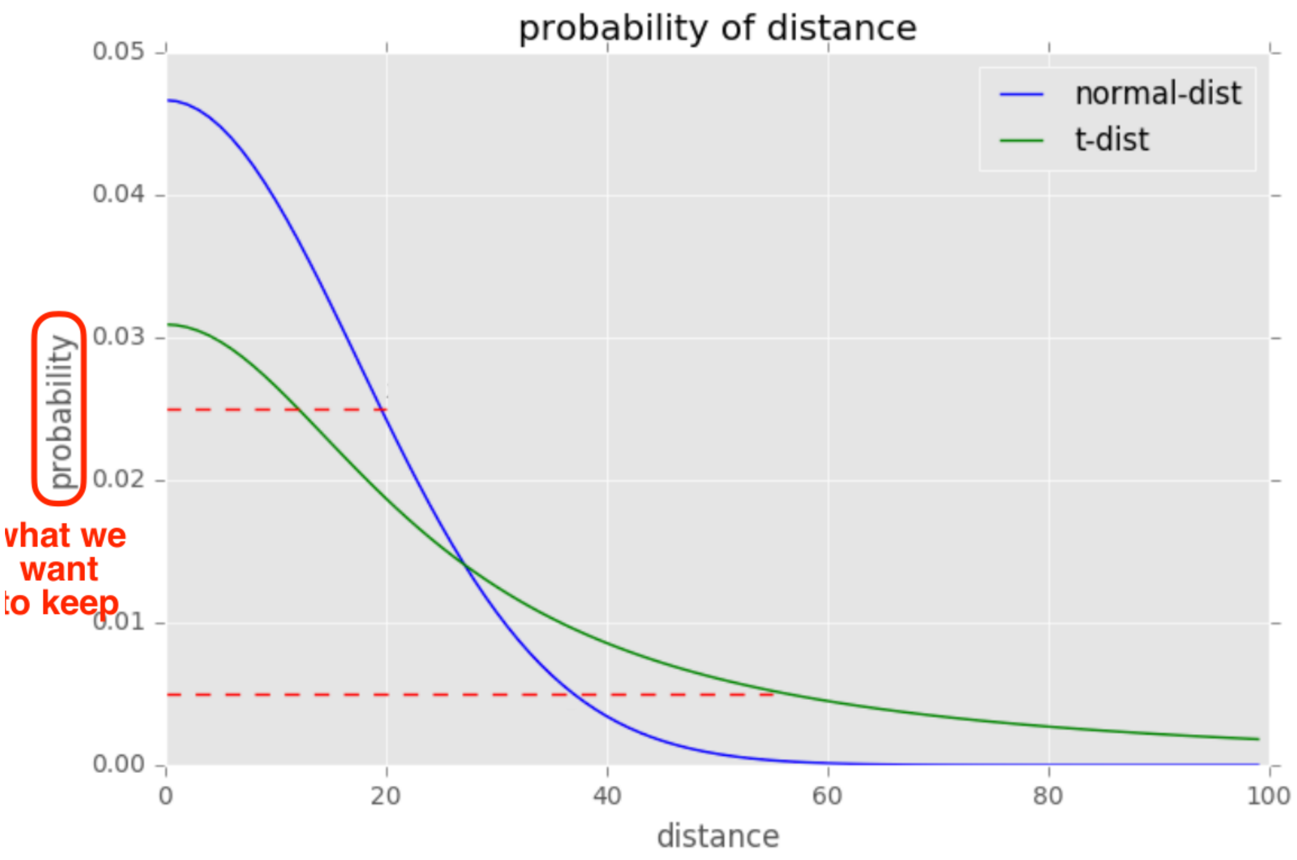


Discussion

t-sne的結果很明顯比較好，原因就是老師上課說的關於sne的crowded problem，如下圖，因為q在分母的關係，所以若q大p小（在High-D距離遠在low-D距離近），給的error相較q小p大（在High-D距離近在low-D距離遠）少很多，因此就會產生這種不管High-D距離遠和近，low-D的距離都很近的情形(crowded problem)。 Crowded problem還有其他原因，以下截自講義。 ‣ in a high-D space, points can have many close-by neighbors in different directions. In a 2D space, you essentially have to arrange close-by neighbors in a circle around the central point, which constrains relationships among neighbors ‣ in a high-D space you can have many points that are equidistant from one another; in 2D, at most 3 points are equidistant from each other ‣ volume of a sphere scales as r d in d dimensions → on a 2D display, there is much less area available at radius r than the corresponding volume in the original space

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

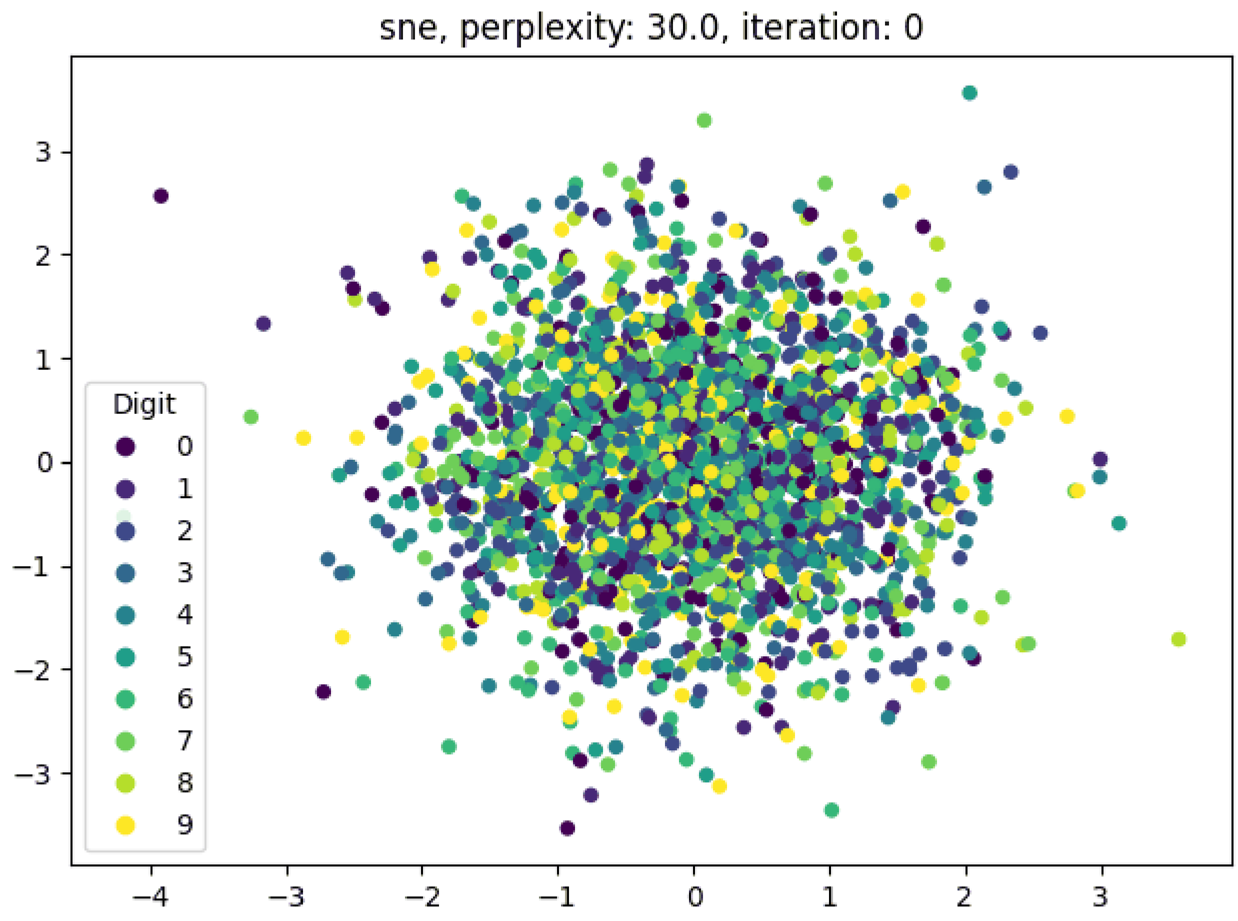
Data在高維空間的體積本來就比較大，硬把它們降到低維度，體積變小，自然就會變得擁擠。 因此，t-sne在低維使用Student-t distribution解決這個問題，Student-T distribution可以把距離相較原本的機率分佈拉遠，見下圖。



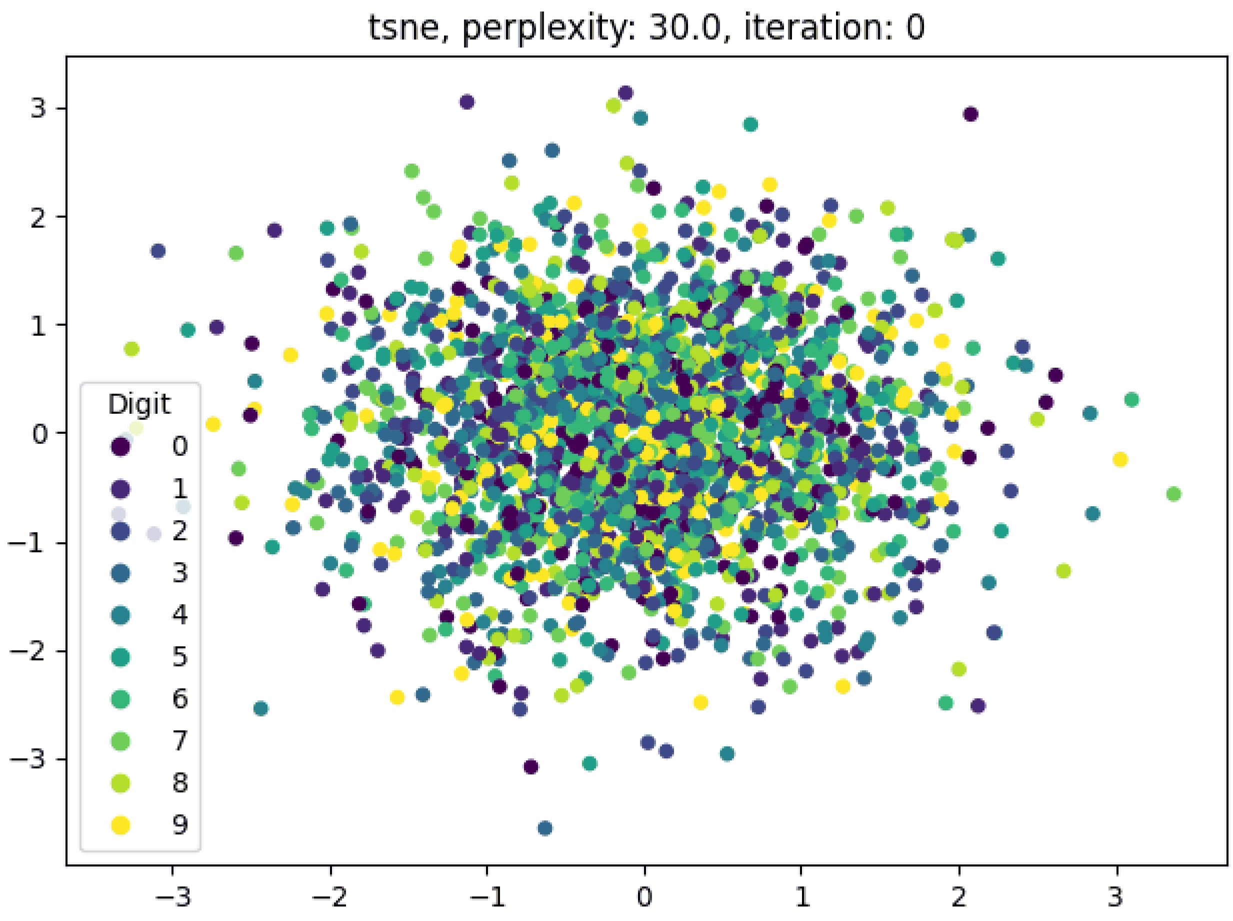
機率較大的(距離近)拉近，機率較小的（距離遠）拉遠，所以可以達到更好的效果。

Part 2

sne

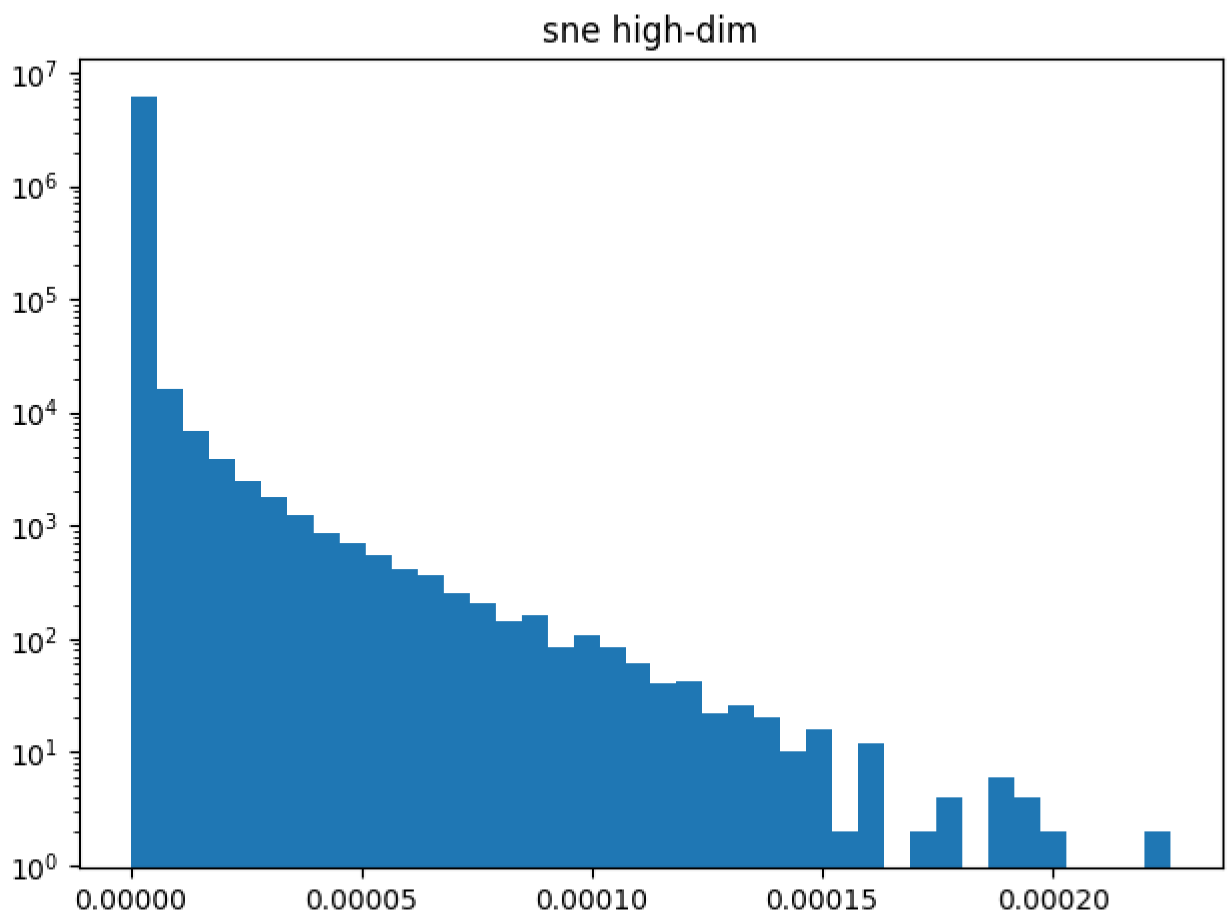

sne, perplexity: 30.0, iteration: 0
Digit
0
1
2
3
4
5
6
7
8
9
3
2
1
0
−1
−2
−3
−4
−3
−2
−1
0
1
2
3

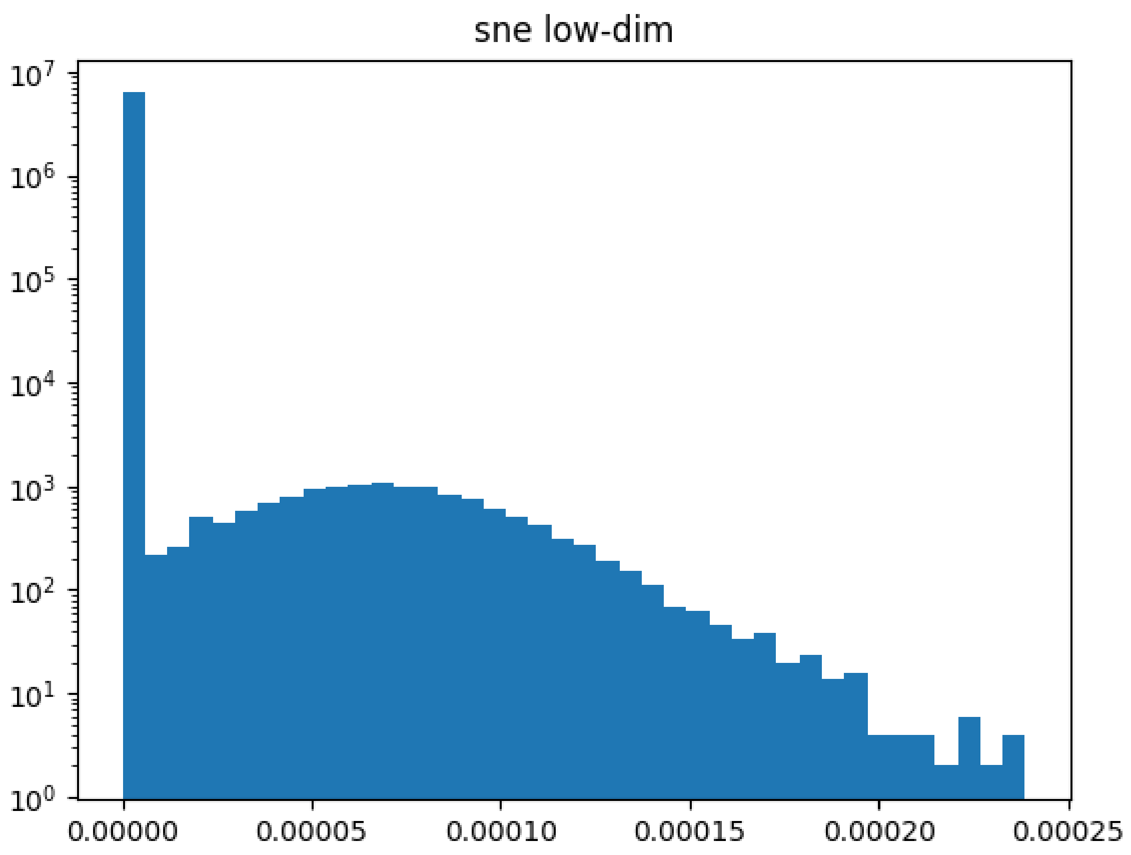
tsne



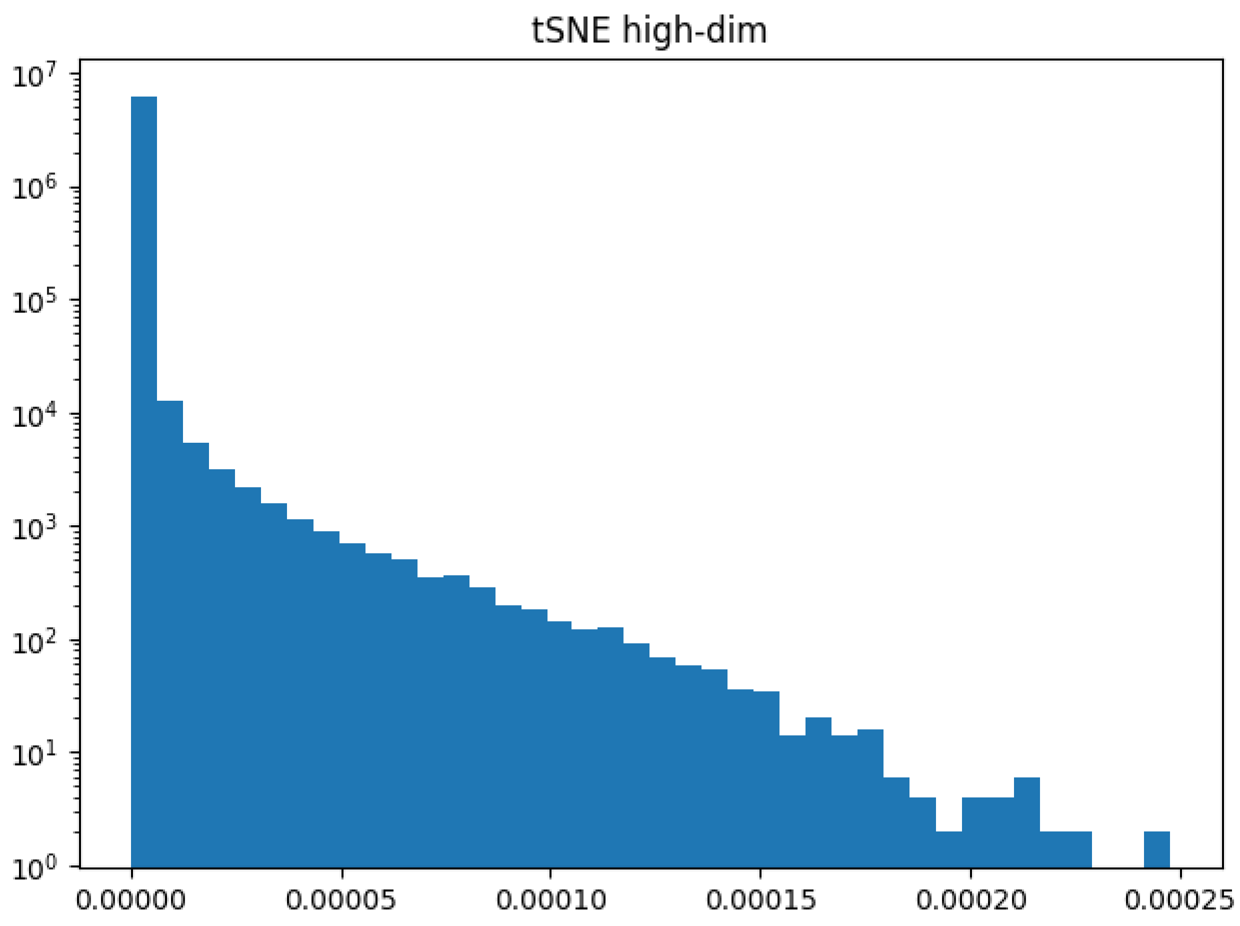
Part 3 Perplexity:30

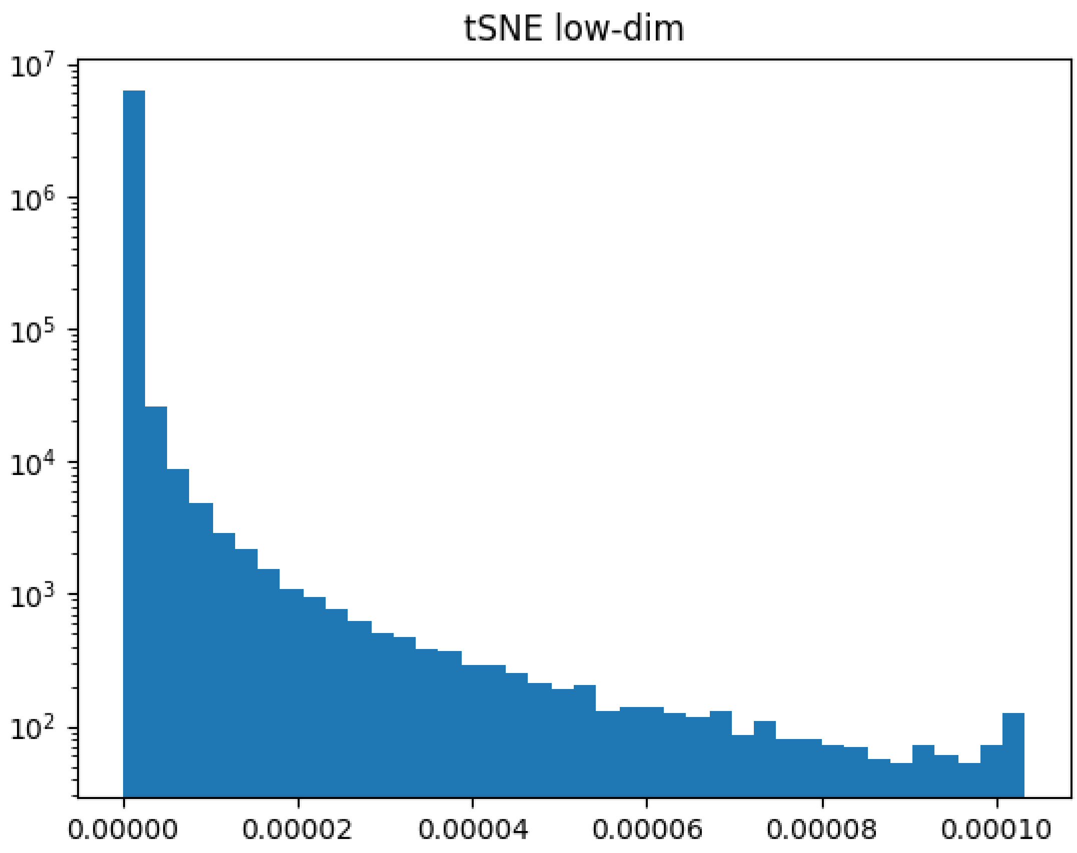
sne


sne high-dim
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0.00000
0.00005
0.00010
0.00015
0.00020


sne low-dim
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025

tsne

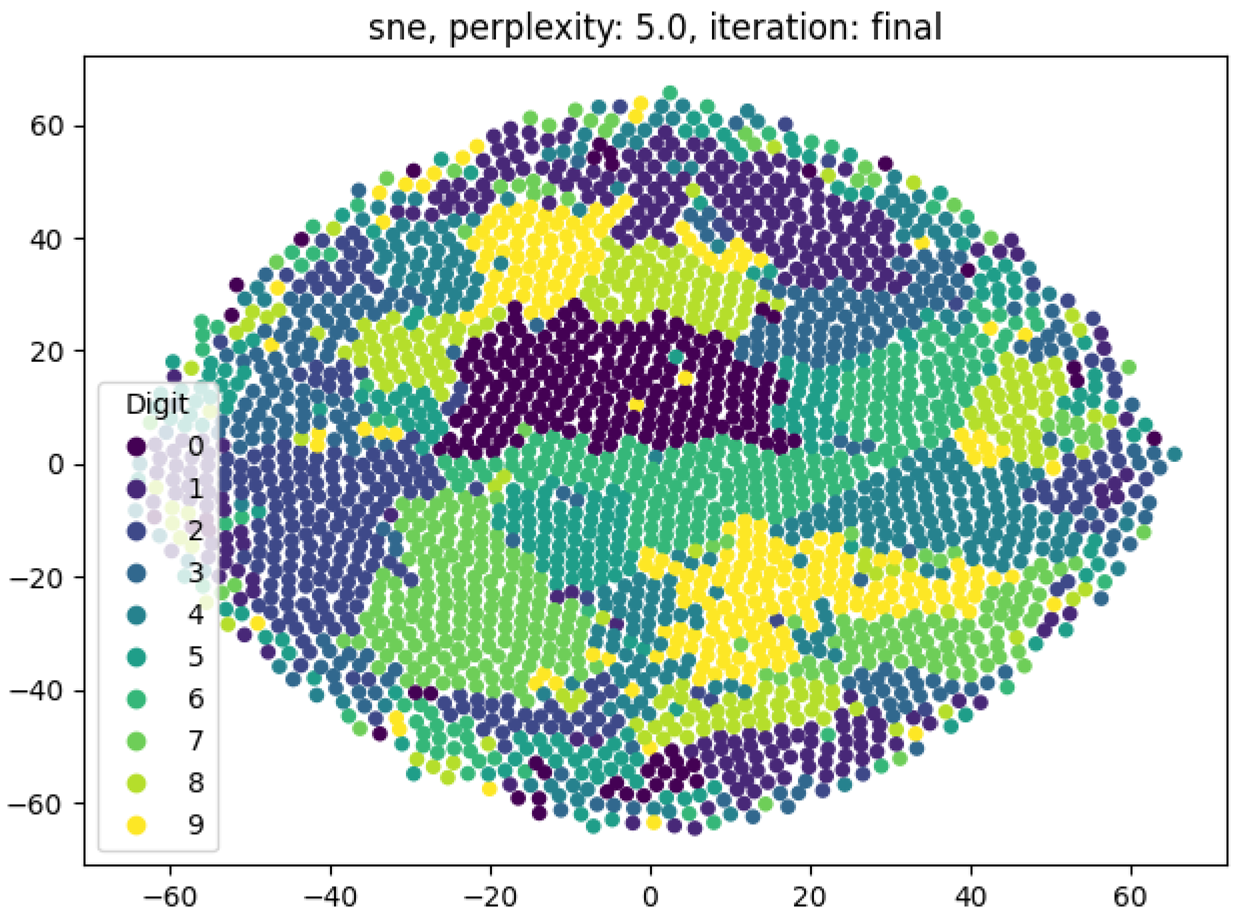

tSNE high-dim
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
0.00000
0.00005
0.00010
0.00015
0.00020
0.00025


tSNE low-dim
10^7
10^6
10^5
10^4
10^3
10^2
0.00000
0.00002
0.00004
0.00006
0.00008
0.00010

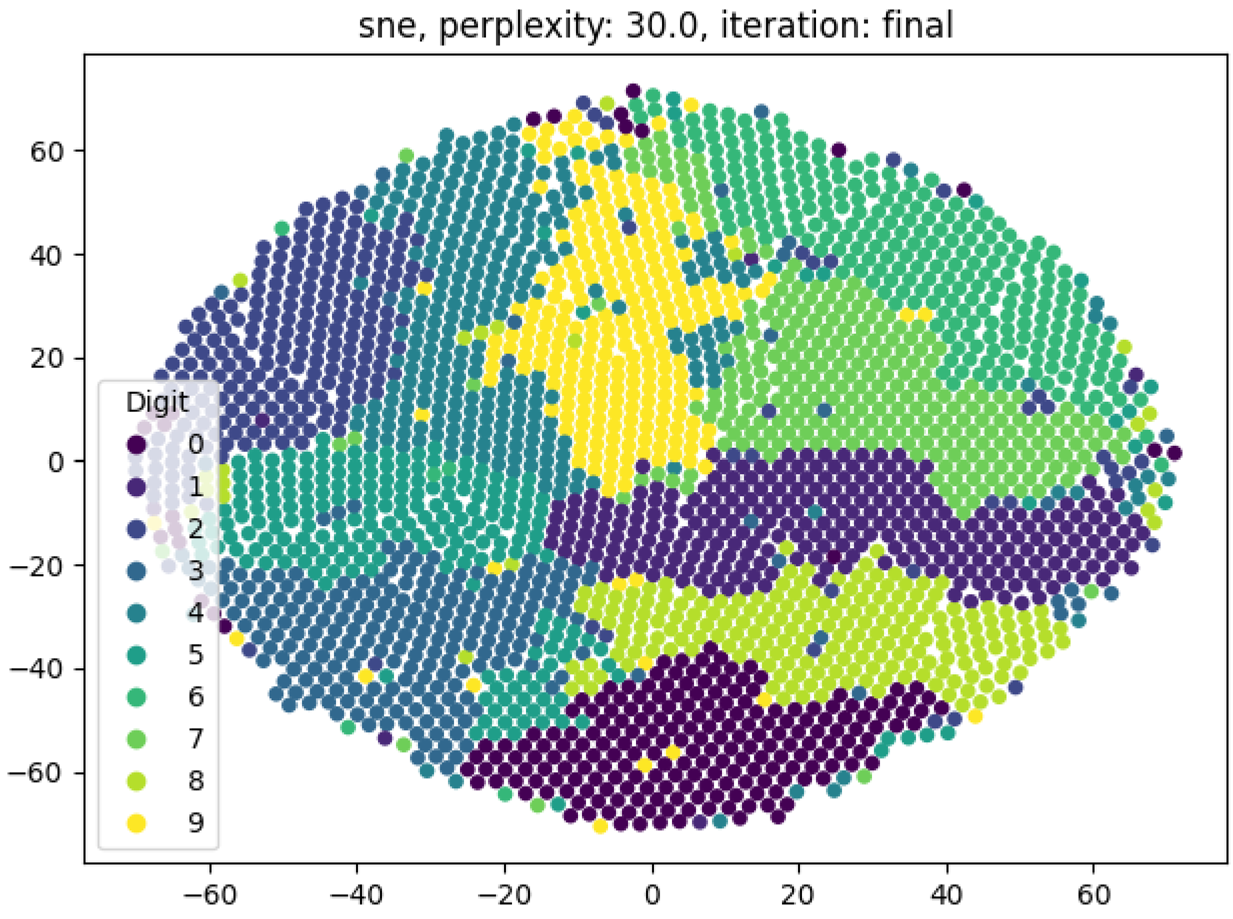
Part 4

sne

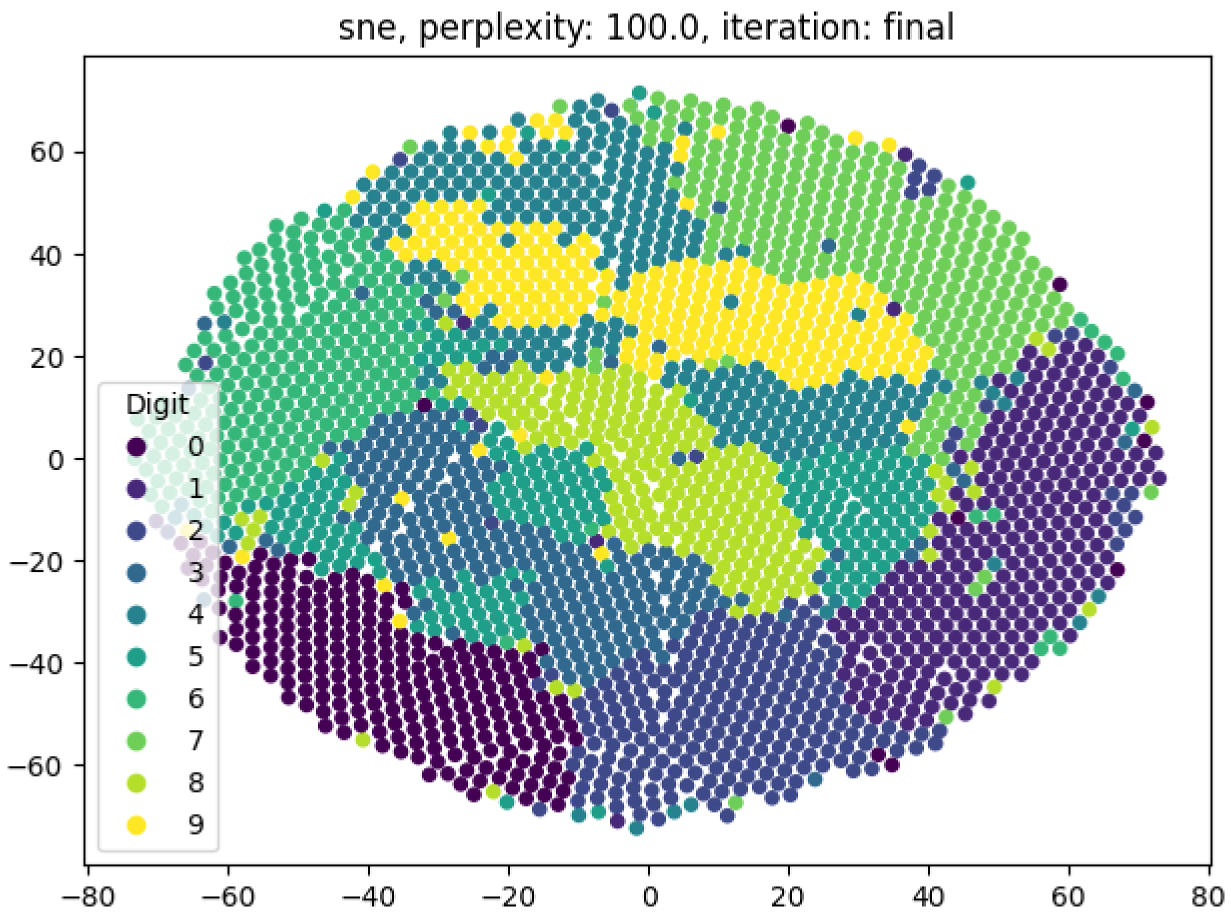
perplexity:5



perplexity:30

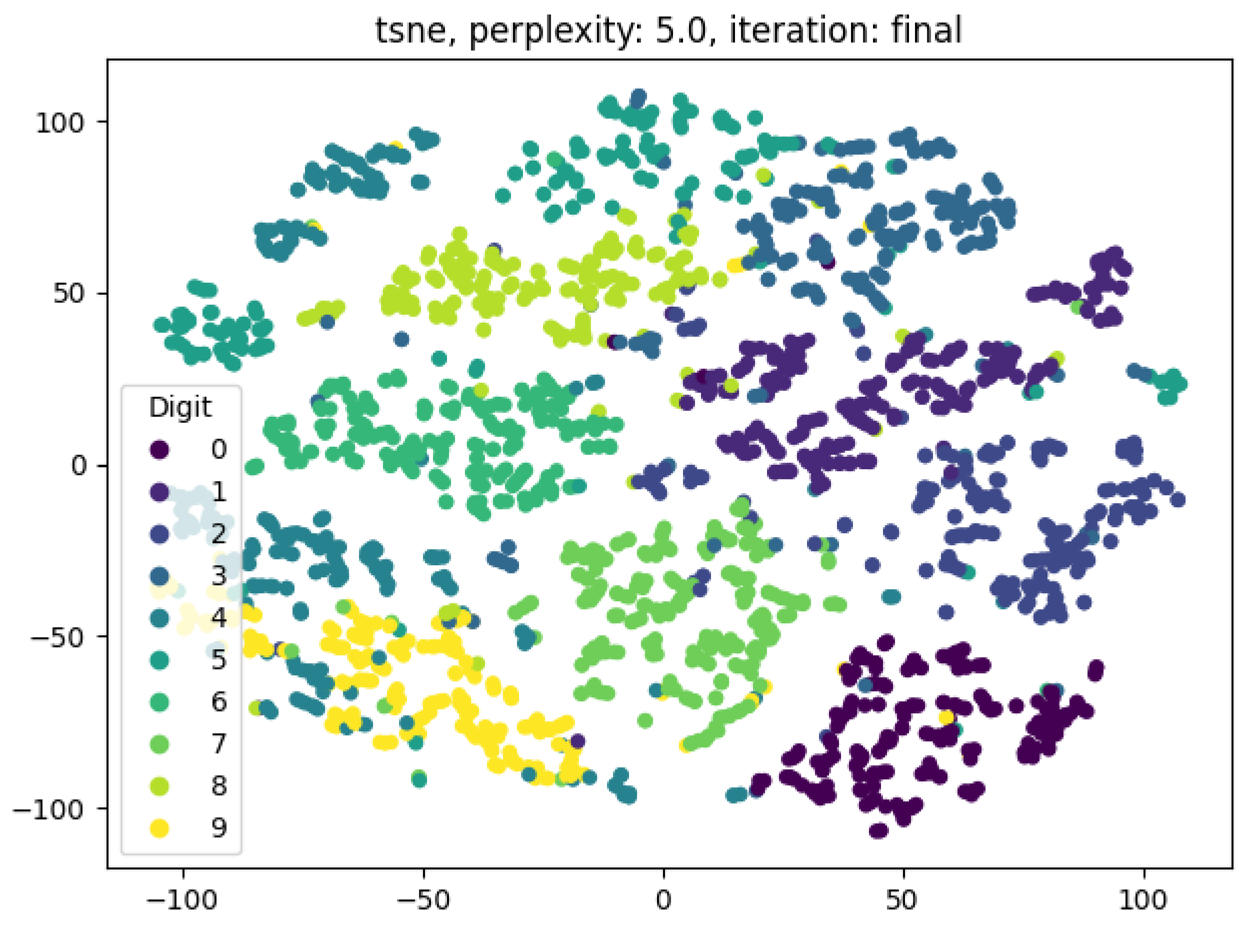


perplexity:100

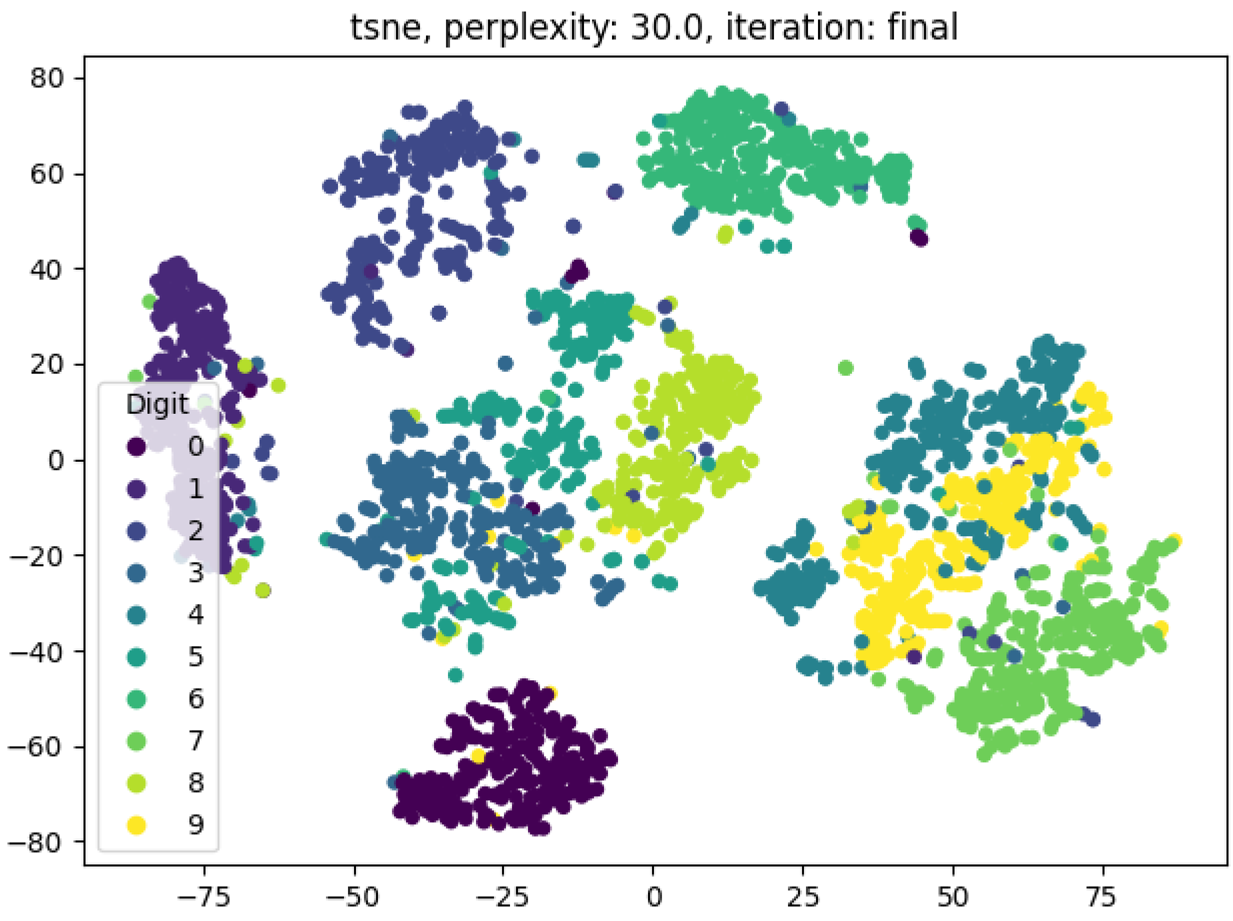


tsne

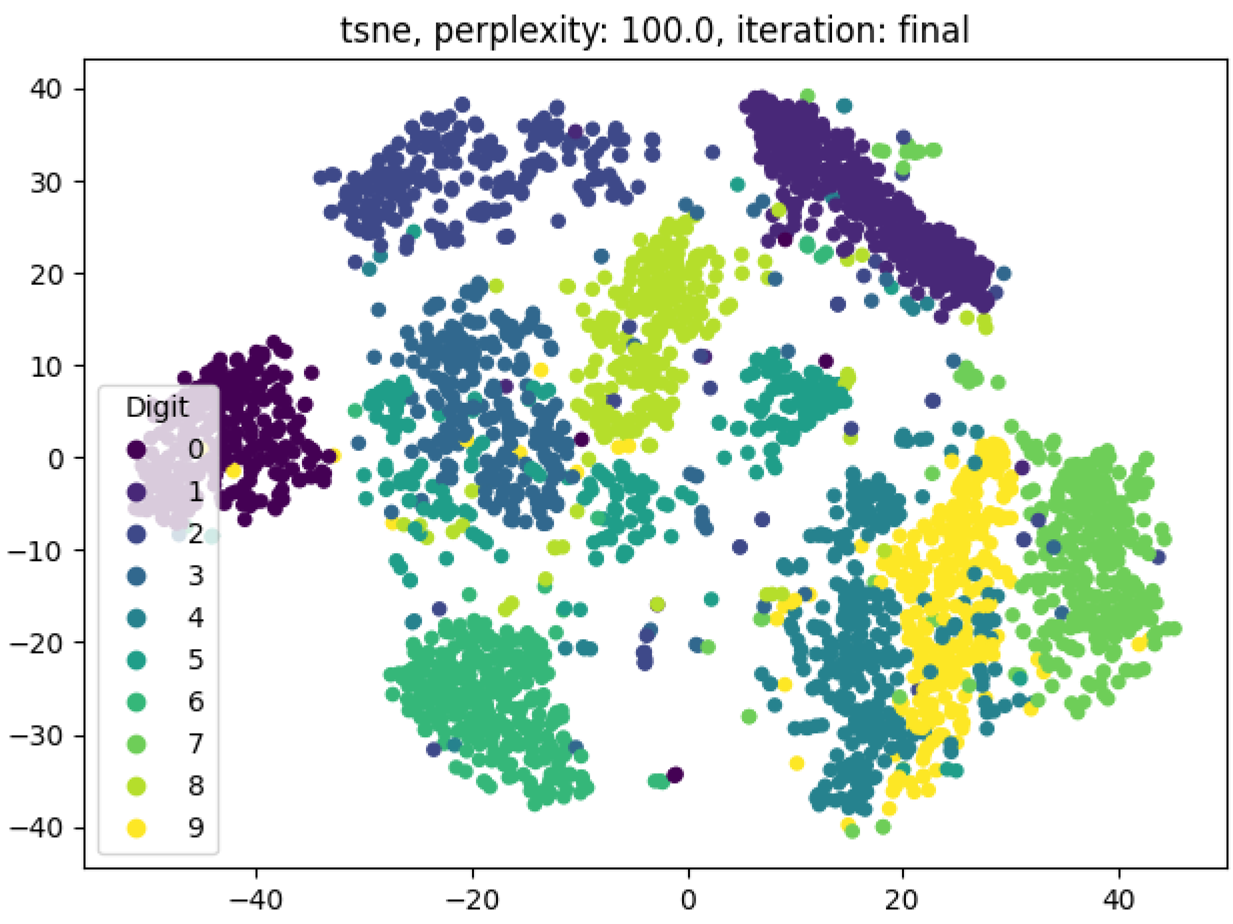
perplexity:5



perplexity:30



perplexity:100



Discussion

通過調整perplexity，我們可以平衡數據的局部和全局方面之間的注意力。對我來說，我認為perplexity意味著每個點的neighbors。我們可以看到 perplexity=5 時，每組中的點定位鬆散。每組被分成幾個小組。當perplexity增加時，每組中的點定位越來越緊密，一組也幾乎都在一起，不再有被分成很多小組的情況。

Summary

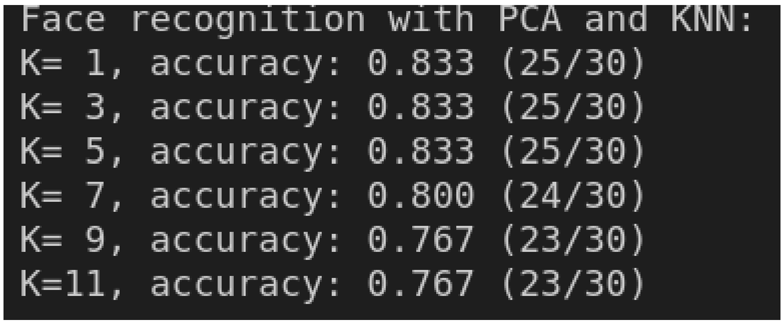
- Perplexity低：只有少數鄰居有影響力，可能把同分群拆成多群。
- Perplexity高：全局結構較明顯，但可能群與群之間會無法區分。

ref : https://mropengate.blogspot.com/2019/06/t-sne.html

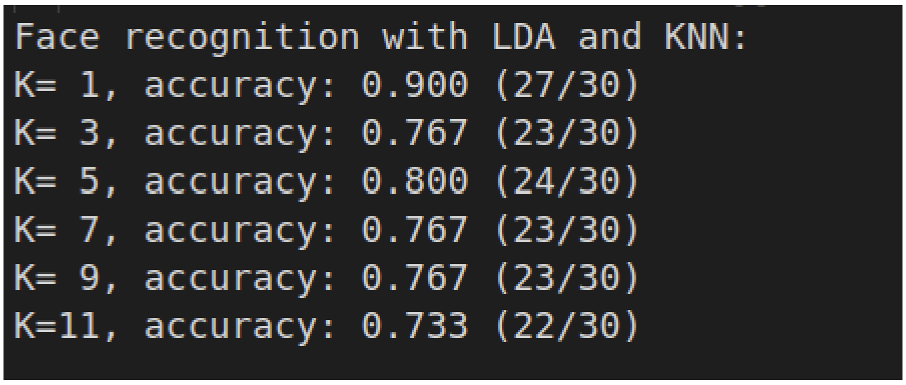
Observations and Discussion

eigenface

LDA的表現會叫較PCA差的原因也可能是因為LDA最多只能降到k-1維（即14維），因為inv(Sw)@SB的rank(SB) < k。因此，PCA扎扎實實取了25維來predict，LDA只有最多14維而已(甚至更少)，以下是降為10維的結果。



```
Face recognition with PCA and KNN:
K= 1, accuracy: 0.833 (25/30)
K= 3, accuracy: 0.833 (25/30)
K= 5, accuracy: 0.833 (25/30)
K= 7, accuracy: 0.800 (24/30)
K= 9, accuracy: 0.767 (23/30)
K=11, accuracy: 0.767 (23/30)
```



```
Face recognition with LDA and KNN:
K= 1, accuracy: 0.900 (27/30)
K= 3, accuracy: 0.767 (23/30)
K= 5, accuracy: 0.800 (24/30)
K= 7, accuracy: 0.767 (23/30)
K= 9, accuracy: 0.767 (23/30)
K=11, accuracy: 0.733 (22/30)
```

PCA的準確度確實下降了，LDA在K=1甚至更好，因此，雖然這次的作業PCA比較好，但以後碰到要做降維分類的case，我應該還是會使用LDA。

t-sne

1. t-SNE 的隨機性

t-SNE 演算法具有隨機性，多次實驗可以產生不同的結果，而一般常見 PCA 是確定性的（deterministic），每次計算後的結果相同。

2. t-SNE 的解釋性

由於演算法本身是基於機率的特性，解釋 t-SNE 的映射時要特別注意：

- 比較兩堆的大小沒有意義：t-SNE 會根據鄰居的擁擠程度來調整區塊的大小
- 比較兩堆間的距離沒有意義：並不是靠比較近的群集彼此就比較像，該嵌入空間中的距離並沒有直覺上的距離的性質。
- t-SNE 不能用於尋找離群點：t-SNE 中的對稱項相當於把離群點拉進某群中。

因此，一般來說只可以用 t-SNE 做定性分析提出假設，不能用 t-SNE 得出結論。 3. 本徵維度（intrinsic dimensionality）

如果不能在 2D 中用 t-SNE 分群，並不一定這個資料就一定不能被模型良好區分，也有可能是 2 維不足夠表示這個資料的內部結構，也就是說原資料的本徵維度過高。

4. 一般 t-SNE 不適用於新資料

PCA 降維可以適用新資料，而一般的 t-SNE 則不行。在 python sklearn library 中的 t-SNE 演算法沒有 transform() 函式。如果真的需要 transform() 方法，需要使用另一種變體：parametric t-SNE，他不包含在 sklearn library 之中。

Barnes-Hut t-SNE

Barnes-Hut t-SNE 是最流行的 t-SNE 方法，主要最佳化 t-SNE 的速度，Barnes-Hut t-SNE 有以下特點：

- 效率提升，Barnes-Hut approximation 使複雜度從 O(n2) 下降至 O(nlogn)，讓 t-SNE 可以處理 10 萬級規模的資料。
- 使用角度（angle）參數對近似結果進行控制。
- 僅在目標維度為 2 或 3 時有效，著重於資料視覺化。
- 稀疏矩陣只能用特定的方法嵌入或者用投影近似。

ref : https://jmlr.org/papers/volume15/vandermaaten14a/vandermaaten14a.pdf