EEC 289A Assignment 1 Report
Chenye Yang, Hanchu Zhou, Haodong Liang, Yibo Ma

1 Introduction

In this project, we aim to do K-mean clustering on the patches from MNIST dataset, shown in Figure 1. After the clustering, we observe the results, and try to answer the following interesting questions:

1. What is the change of the learned clusters when K increases from 100 - 10,000?
2. How well one can reconstruct a 5x5 MNIST patch by the learned dictionary (clusters)?
3. How many clusters does one need in order to cover the whole patch space?
4. What are these clusters and do they have any interpretable meanings?
5. How is one digit made from these clusters?

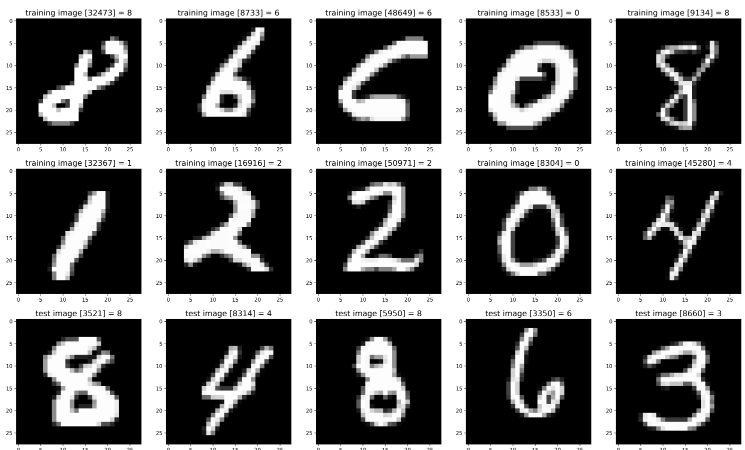


Figure 1: Some examples of the MNIST dataset

2 Methodology

2.1 Data Preprocessing

We load the MNIST dataset (28×28 handwritten digits)[1], including 60000 training images and 10000 testing images. The pixel values in each image are not normalized, and they range from 0 (black) to 255 (white). Then, we extract the patches from the training images, by sliding a 5×5 window over the images. Therefore, for each handwritten digit, we will get $(28-5+1)\times(28-5+1)$ patches. Then we get rid of all the blank patches, leading to 20,074,704 non-blank patches in total. Each patch is reshaped to a 25-dimensional vector, and we get a matrix $X \in R^{20,074,704\times25}$.

2.2 K-mean clustering

Once we get all the patches, we can do the K-mean clustering on the patches. The K-mean clustering is a method to partition the data into K clusters, where each data point belongs to the cluster with the nearest mean. Although there is a wildly used K-means algorithm in library *scikit learn*[2], we also implement the

[1] Downloaded from `https://git-disl.github.io/GTDLBench/datasets/mnist_datasets/`
[2] `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`

K-means clustering algorithm using PyTorch, hoping to achieve acceleration by using Nvidia CUDA[3] or Apple MPS[4], which is shown in the Algorithm 1.

We define that:

- K: Number of the clustering
- C: Centroids of the clustering
- P: The 5×5 patch
- X: The entire dataset of patches with size $20,074,704 \times 25$

Algorithm 1 PyTorch K-means Clustering for 5×5 Patches

procedure KMeansPatches($Patches, K$)
    $X \leftarrow$ Reshape each patch P in $Patches$ to R^{25} ▷ No normalization
    Initialize C random K data points from X
    $C_{\text{old}} \leftarrow$ Copy of C
    repeat
        Compute distances from each vector X_i in X to each centroid in C
        Assign each vector X_i to the closest centroid
        for $j \leftarrow 1$ to K **do**
            if Count(X_i assigned to C_j) $= 0$ **then**
                Randomly reinitialize centroid C_j from X
            else
                Update centroid C_j by calculating the mean of all vectors assigned to C_j
            end if
        end for
        $C_{\text{new}} \leftarrow$ Copy of C
        $C_{\text{move}} \leftarrow \text{norm}(C_{\text{new}} - C_{\text{old}})$
        $C_{\text{old}} \leftarrow C_{\text{new}}$
    until $C_{\text{move}} <$ tolerance
    return Updated centroids and cluster labels
end procedure

2.3 Reconstruction

After the clustering, we can use the centroids to reconstruct the original digits, as shown in Algorithm 2. The reconstruction is done by assigning each non-blank patch to the nearest centroid, and keep the blank patches as zeros. Because of the overlap of the patches, we will have multiple centroids assigned to the same pixel. Thus, we need a count matrix to record the number of how many centroids are assigned to each pixel. Finally, we average the value of each pixel to get the reconstructed digit.

Some notations in this algorithm:

- POS: The position indices of a specific patch
- D: The ground truth of the handwritten digit
- $\hat{D}$: The reconstructed result of D according to the K-mean clustering centroids

[3] `https://pytorch.org/docs/stable/cuda.html`
[4] `https://developer.apple.com/metal/pytorch/`

Algorithm 2 Reconstruct Handwritten Digit Images

```
procedure ReconstructDigit(D, K, Model)
    P ← empty list to store patches
    Pos ← empty list to store position indices
    for i ← 1 to 24 do                                          ▷ 28 - 5 + 1 = 24
        for j ← 1 to 24 do
            patch ← D[i : i + 5, j : j + 5]
            if not all zeros in patch then
                P.append(patch.flatten())
                Pos.append((i, j))
            end if
        end for
    end for
    X ← stack of patches in P
    Labels ← Model.predict(X)                                   ▷ Assign patches to centroids
    D̂ ← zero matrix of the same size as D                      ▷ Initialize reconstructed digit with zeros
    Count ← zero matrix of the same size as D                   ▷ Initialize Count matrix with zeros
    for k ← 0 to len(P) − 1 do
        pos ← Pos[k]
        cluster ← Labels[k]
        centroid ← Model.cluster_centers[cluster].reshape(5, 5)
        D̂[pos[0] : pos[0] + 5, pos[1] : pos[1] + 5] ← D̂[pos[0] : pos[0] + 5, pos[1] : pos[1] + 5] + centroid
        Count[pos[0] : pos[0] + 5, pos[1] : pos[1] + 5] ← Count[pos[0] : pos[0] + 5, pos[1] : pos[1] + 5] + 1
    end for
    Count[Count == 0] = 1                                       ▷ Avoid division by 0
    return D̂/Count
end procedure
```

3 Experiment Results

3.1 Clustering Results with Different K

We conducted the K-mean clustering on the patches with different K values, where:

$$K = 100, 200, 300, \ldots, 900, 1000, 2000, \ldots, 9000, 10000.$$

Some results of the patches and the corresponding centroids are shown in the following Figures 2-7.

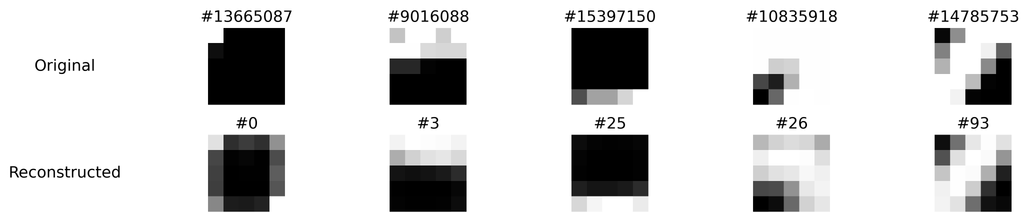


Figure 2: The patch and corresponding centroid when $K = 100$

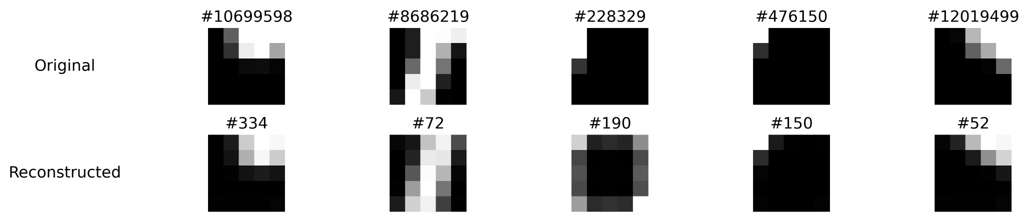


Figure 3: The patch and corresponding centroid when $K = 400$

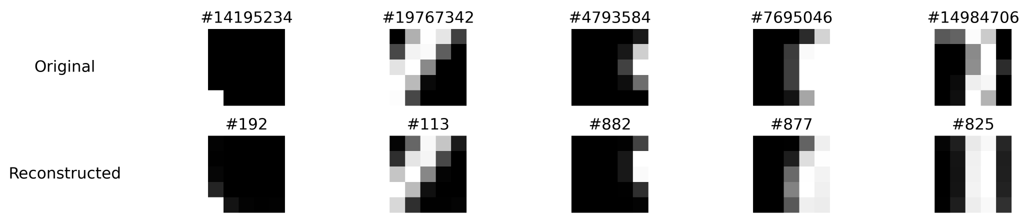


Figure 4: The patch and corresponding centroid when $K = 1000$

Figure 5: The patch and corresponding centroid when $K = 2000$

Moreover, we plot the Mean Squared Error (MSE) between the original patches and the reconstructed patches (centroids) with different K values, as shown in Figure 8.

With the results above, we can observe that:

Figure 6: The patch and corresponding centroid when $K = 5000$

Figure 7: The patch and corresponding centroid when $K = 10000$

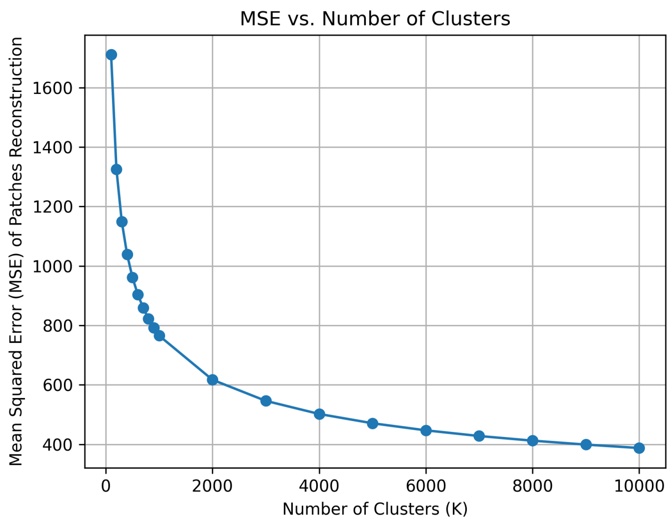
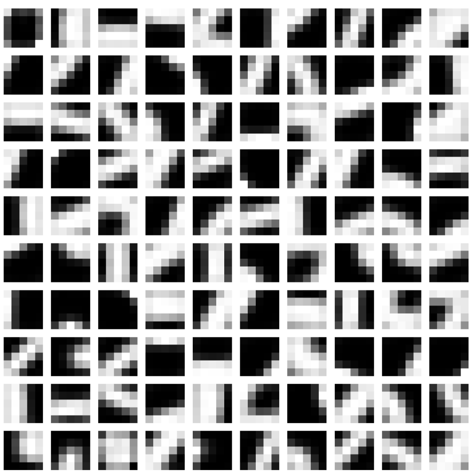


Figure 8: MSE with different clustering number

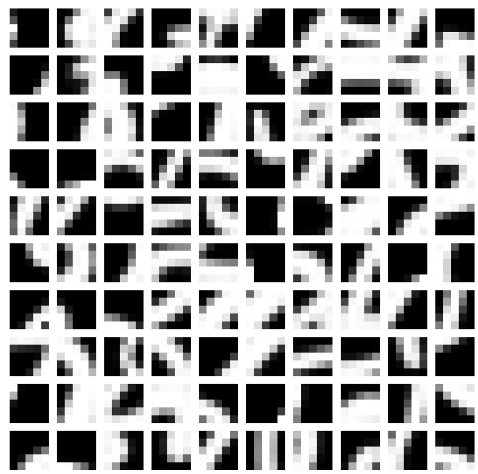
- Question 1: As K increases, the learned clusters become more and more detailed and similar to the original patches.
- Question 2: With the increase of K, one can reconstruct the patches more accurately. For example, when $K = 100$, the reconstructed patches are not very similar, but when $K = 10000$, the reconstructed patches are very similar to the original patches.
- Question 3: The MSE decreases as K increases, however, the decreasing rate becomes slower when K is large. By elbow method, we can find that the optimal K is around 2000. So we need 2000 clusters to cover the whole patch space, considering the trade-off between the clustering performance and the computational cost.

3.2 What are the centroids

In addition to the clustering performance, we also want to understand the meaning of the clusters or centroids. Here we show the first 100 centroids when using $K = 100$ and $K = 10000$ in Figure 9.



(a) $K = 100$



(b) $K = 10000$

Figure 9: Visual representation of cluster centroids for $K = 100$ and $K = 10000$

Question 4: From the results, we can see that the centroids of these clusters are actually the patterns of the handwritten digits. Specifically, they are the edges, corners, and other features of the digits.

3.3 Reconstruct a Digit

In this part, we show the reconstruction results of the handwritten digits from both the training set and testing set, by using the learned centroids, as in Figures 10 and 11.

From the results, we can observe that the reconstructed digits are very vague when $K = 100$, but they become more and more clear as K increases. When $K = 10000$, the reconstructed digits are very similar to the original digits. Question 5: This indicates that the digits are constructed from the detailed features of itself, such as the edges, corners, and other features. One digit is just a linear combination of these features / centroids/ clusters, and the more clusters we have, the more accurate the reconstruction will be.

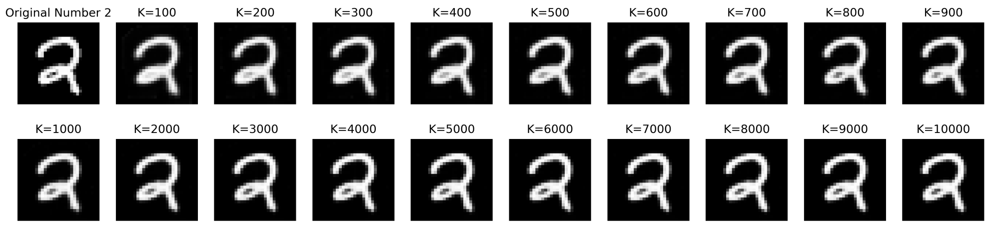


Figure 10: Reconstruction of digit 2 with different K

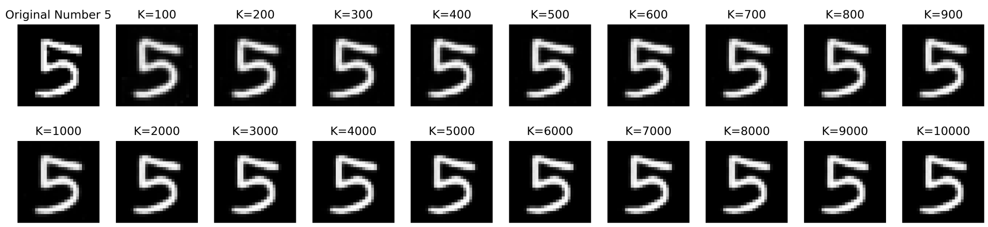


Figure 11: Reconstruction of digit 5 with different K

Appendix

```
# https://www.kaggle.com/code/hojjatk/read-mnist-dataset/notebook
#
# This is a sample Notebook to demonstrate how to read "MNIST Dataset"
#
import numpy as np # linear algebra
import struct
from array import array

#
# MNIST Data Loader Class
#
class MnistDataloader(object):
    '''
    MNIST Data Loader

        @type   training_images_filepath: string
        @param  training_images_filepath: training images file path

        @type   training_labels_filepath: string
        @param  training_labels_filepath: training labels file path

        @type   test_images_filepath: string
        @param  test_images_filepath: test images file path

        @type   test_labels_filepath: string
        @param  test_labels_filepath: test labels file path
    '''
```

```
    def __init__(self, training_images_filepath, training_labels_filepath, test_images_filepath, test_labels_filepath):
        self.training_images_filepath = training_images_filepath
        self.training_labels_filepath = training_labels_filepath
        self.test_images_filepath = test_images_filepath
        self.test_labels_filepath = test_labels_filepath

    def read_images_labels(self, images_filepath, labels_filepath):
        '''
        Read images and labels

                @type    images_filepath: string
                @param   images_filepath: images file path

                @type    labels_filepath: string
                @param   labels_filepath: labels file path

                @rtype:    (ndarray, ndarray)
                @return:   images, labels
        '''
        labels = []
        with open(labels_filepath, 'rb') as file:
            magic, size = struct.unpack(">II", file.read(8))
            if magic != 2049:
                raise ValueError('Magic number mismatch, expected 2049, got {}'.format(magic))
            labels = array("B", file.read())

        with open(images_filepath, 'rb') as file:
            magic, size, rows, cols = struct.unpack(">IIII", file.read(16))
            if magic != 2051:
                raise ValueError('Magic number mismatch, expected 2051, got {}'.format(magic))
            image_data = array("B", file.read())
        images = []
        for i in range(size):
            images.append([0] * rows * cols)
        for i in range(size):
            img = np.array(image_data[i * rows * cols:(i + 1) * rows * cols])
            img = img.reshape(28, 28)
            images[i][:] = img

        return images, labels

    def load_data(self):
        '''
        Load MNIST data

            @rtype:    (ndarray, ndarray), (ndarray, ndarray)
            @return:   (training data, traing labels), (test data, test labels)
        '''
        x_train, y_train = self.read_images_labels(self.training_images_filepath, self.training_labels_filepath)
```

```python
        x_test, y_test = self.read_images_labels(self.test_images_filepath, self.test_labels_filepath)
        return (np.array(x_train), np.array(y_train)),(np.array(x_test), np.array(y_test))
```

```python
from mnist_data_loader import MnistDataloader
from os.path  import join
import numpy as np
from tqdm import tqdm


# Extract 5x5 patches from the 28x28 images
def extract_patches(images, patch_size=5, threshold=0):
    '''
    Extract patches from images

        @type   images: ndarray
        @param  images: images

        @type   patch_size: int
        @param  patch_size: patch size

        @type   threshold: float
        @param  threshold: default 0 means non-blank patches from the training images

        @rtype:   ndarray
        @return:  patches
    '''
    patches = []
    num_pixels = patch_size * patch_size
    for image in tqdm(images, desc="Extracting patches"):
        # Slide over the image and extract patches
        for i in range(image.shape[0] - patch_size + 1):
            for j in range(image.shape[1] - patch_size + 1):
                patch = image[i:i + patch_size, j:j + patch_size]
                # Calculate the proportion of non-zero pixels
                if np.sum(patch) > 255 * num_pixels * threshold:  # Adjust the threshold as needed
                    patches.append(patch.flatten())
    return np.array(patches)


# Extract 5x5 patches from the 28x28 images
def extract_nonblank_patches_from_one_image(image, patch_size=5, threshold=0):
    '''
    Extract nonblank patches from one image

        @type   image: ndarray
        @param  image: image

        @type   patch_size: int
        @param  patch_size: patch size
```

```
        @type    threshold: float
        @param   threshold: default 0 means non-blank patches from the training images

        @rtype:    ndarray
        @return:   patches
    '''
    patches = []
    num_pixels = patch_size * patch_size
    # Slide over the image and extract patches
    for i in range(image.shape[0] - patch_size + 1):
        for j in range(image.shape[1] - patch_size + 1):
            patch = image[i:i + patch_size, j:j + patch_size]
            # Calculate the proportion of non-zero pixels
            if np.sum(patch) > 255 * num_pixels * threshold:  # Adjust the threshold as needed
                patches.append(patch.flatten())
    return np.array(patches)


def extract_all_patches_from_one_image(image, patch_size=5):
    '''
    Extract all patches (blank and nonblank) from one image

        @type    image: ndarray
        @param   image: image

        @type    patch_size: int
        @param   patch_size: patch size

        @rtype:    ndarray
        @return:   patches
    '''
    patches = []
    # Slide over the image and extract patches
    for i in range(image.shape[0] - patch_size + 1):
        for j in range(image.shape[1] - patch_size + 1):
            patch = image[i:i + patch_size, j:j + patch_size]
            patches.append(patch.flatten())
    return np.array(patches)



if __name__ == "__main__":
    # Set file paths based on added MNIST Datasets
    input_path = 'MNIST_ORG/'
    training_images_filepath = join(input_path, 'train-images.idx3-ubyte')
    training_labels_filepath = join(input_path, 'train-labels.idx1-ubyte')
    test_images_filepath = join(input_path, 't10k-images.idx3-ubyte')
    test_labels_filepath = join(input_path, 't10k-labels.idx1-ubyte')

    # Load MINST dataset
    mnist_dataloader = MnistDataloader(training_images_filepath,
```

```
        training_labels_filepath, test_images_filepath, test_labels_filepath)
    (x_train, y_train), (x_test, y_test) = mnist_dataloader.load_data()


    # Extract non-blank patches from the training data
    patches = extract_patches(x_train)
    np.save("patches.npy", patches)
```

```
import torch
from tqdm import tqdm

def kmeans_pytorch(X, n_clusters, n_iters=100, tolerance=1e-4):
    """
    Performs k-means clustering using PyTorch.

    Parameters:
        X (torch.Tensor): The input data, a tensor of shape (n_samples,
            n_features).
        n_clusters (int): The number of clusters to form.
        n_iters (int): Maximum number of iterations of the k-means algorithm.
        tolerance (float): Tolerance to declare convergence.

    Returns:
        centers (torch.Tensor): Cluster centers, a tensor of shape (n_clusters
            , n_features).
        labels (torch.Tensor): Index of the cluster each sample belongs to.
    """
    # Randomly choose cluster centers from the input data at the start.
    indices = torch.randperm(X.size(0))[:n_clusters]
    centers = X[indices]

    for _ in tqdm(range(n_iters), desc="K-means"):
        # Compute distances from data points to the centroids
        distances = torch.cdist(X, centers)
        # Assign clusters
        labels = torch.argmin(distances, dim=1)
        # Compute new centers
        new_centers = torch.stack([X[labels == i].mean(dim=0) for i in range(
            n_clusters)])

        # Check for convergence
        if torch.norm(centers - new_centers) < tolerance:
            break

        centers = new_centers

    return centers, labels


if __name__ == "__main__":
    # Example usage
    # Creating some data
    torch.manual_seed(0)
```

```
    data = torch.randn(100, 2)  # 100 data points, 2 dimensions

    # Clustering
    centers, labels = kmeans_pytorch(data, n_clusters=3)
    print("Cluster centers:\n", centers)
    print("Cluster labels:\n", labels)
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import joblib
from tqdm import tqdm

# Load patches
try:
    patches = np.load("K-means/Code/patches.npy")
except FileNotFoundError:
    print("cd to folder EEC289A, run extract_patches.py first.")
    exit()

K = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000,
    5000, 6000, 7000, 8000, 9000, 10000]

# Perform K-means clustering
for n_clusters in tqdm(K, desc="Clustering"):
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(patches)

    # Save the model
    joblib.dump(kmeans, f"K-means/Result/Model/{n_clusters}-clusters-model.
        joblib")


# # Load the model
# model = joblib.load("../Result/Model/100-clusters-model.joblib")
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import mean_squared_error
import joblib


def visualize_reconstruction(n_clusters, patches, model):
    '''
    Visualize the original and reconstructed patches for a few random samples.

        @type    n_clusters: int
        @param   n_clusters: K

        @type    patches: ndarray
        @param   patches: patches

        @type    model: sklearn model
```

```python
        @param  model: the fitted sklearn KMeans model
    '''
    # Predict the cluster for each patch
    labels = model.labels_

    # Get the cluster centers
    centroids = model.cluster_centers_

    # Pick random patches for display
    num_samples = 5  # Number of random samples to pick
    indices = np.random.choice(range(len(patches)), num_samples, replace=False)

    # Plotting the original and reconstructed patches
    fig, axs = plt.subplots(2, num_samples+1, figsize=(15, 3))  # 2 rows: originals and reconstructions

    # Set labels for the rows
    axs[0, 0].text(0.5, 0.5, 'Original', verticalalignment='center', horizontalalignment='center', transform=axs[0, 0].transAxes, fontsize = 15)
    axs[1, 0].text(0.5, 0.5, 'Reconstructed', verticalalignment='center', horizontalalignment='center', transform=axs[1, 0].transAxes, fontsize = 15)
    axs[0, 0].axis('off')
    axs[1, 0].axis('off')

    for i, idx in enumerate(indices):
        i += 1  # Adjust index for the extra label column

        # Original patches
        axs[0, i].imshow(patches[idx].reshape(5, 5), cmap='gray')
        axs[0, i].axis('off')
        axs[0, i].set_title('#{}'.format(idx), fontsize = 15)

        # Reconstructed patches
        reconstructed_patch = centroids[labels[idx]].reshape(5, 5)
        axs[1, i].imshow(reconstructed_patch, cmap='gray')
        axs[1, i].axis('off')
        axs[1, i].set_title('#{}'.format(labels[idx]), fontsize = 15)

    plt.tight_layout()
    # plt.show()
    plt.savefig(f"K-means/Result/Reconstruction/{n_clusters}-clusters-reconstruction.png", dpi=300)
    plt.close()


def mse_reconstruction(patches, model):
    '''
    Calculate the mean squared error between the original and reconstructed patches.

        @type   patches: ndarray
```

```python
        @param  patches: patches

        @type   model: sklearn model
        @param  model: the fitted sklearn KMeans model
    '''
    # Predict the cluster for each patch
    labels = model.labels_

    # Get the cluster centers
    centroids = model.cluster_centers_

    # Calculate the mean squared error
    reconstruction = centroids[labels]
    mse = mean_squared_error(patches, reconstruction)

    return mse




if __name__ == "__main__":
    K = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000,
        5000, 6000, 7000, 8000, 9000, 10000]

    # Load patches
    try:
        patches = np.load("K-means/Code/patches.npy")
    except FileNotFoundError:
        print("cd to folder EEC289A, run extract_patches.py first.")
        exit()

    mse_K = []

    for n_clusters in K:
        # Load the model
        try:
            model = joblib.load(f"K-means/Result/Model/{n_clusters}-clusters-
                model.joblib")
        except FileNotFoundError:
            print(f"cd to folder EEC289A, run run_kmeans.py first.")
            exit()

        visualize_reconstruction(n_clusters, patches, model)

        mse = mse_reconstruction(patches, model)
        mse_K.append(mse)

    # Plot the mean squared error
    plt.plot(K, mse_K, marker='o')
    plt.xlabel('Number of Clusters (K)')
    plt.ylabel('Mean Squared Error (MSE) of Patches Reconstruction')
    plt.title('MSE vs. Number of Clusters')
    plt.grid(True)
```

```
    plt.savefig("K-means/Result/Reconstruction/MSE_vs_K.png", dpi=300)
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import joblib
from os.path  import join

from mnist_data_loader import MnistDataloader
from extract_patches import extract_all_patches_from_one_image,
    extract_nonblank_patches_from_one_image



def calculate_positions(image_shape, patch_size):
    '''
    Calculate the positions of all patches in an image

        @type   image_shape: tuple
        @param  image_shape: image shape

        @type   patch_size: int
        @param  patch_size: patch size

        @rtype:    list
        @return:   positions
    '''
    positions = []
    for i in range(image_shape[0] - patch_size + 1):
        for j in range(image_shape[1] - patch_size + 1):
            positions.append((i, j))
    return positions



def reconstruct_digit(digit_image, model, patch_size = 5):
    '''
    Reconstruct a digit image using a KMeans model

        @type   digit_image: ndarray
        @param  digit_image: digit image

        @type   model: sklearn model
        @param  model: KMeans model

        @type   patch_size: int
        @param  patch_size: patch size

        @rtype:   ndarray
        @return:  reconstructed image
    '''
    # Extract all patches and calculate positions
    all_patches = extract_all_patches_from_one_image(digit_image)
```

```python
    positions = calculate_positions(digit_image.shape, patch_size)

    # Extract non-blank patches
    nonblank_patches = extract_nonblank_patches_from_one_image(digit_image)

    # Map nonblank patches to their positions
    nonblank_indices = [i for i, patch in enumerate(all_patches) if np.sum(patch) > 0]

    # Predict clusters for non-blank patches
    labels = model.predict(nonblank_patches)

    # Initialize the reconstructed image with zeros
    reconstructed_image = np.zeros_like(digit_image, dtype=float)
    count_matrix = np.zeros_like(digit_image, dtype=float)

    # Add centroids to the corresponding positions
    centroids = model.cluster_centers_
    for label, idx in zip(labels, nonblank_indices):
        i, j = positions[idx]
        reconstructed_image[i:i+patch_size, j:j+patch_size] += centroids[label].reshape(patch_size, patch_size)
        count_matrix[i:i+patch_size, j:j+patch_size] += 1

    # Avoid division by zero
    count_matrix[count_matrix == 0] = 1
    reconstructed_image /= count_matrix

    return reconstructed_image





if __name__ == "__main__":
    # Set file paths based on added MNIST Datasets
    input_path = 'K-means/Code/MNIST_ORG/'
    training_images_filepath = join(input_path, 'train-images.idx3-ubyte')
    training_labels_filepath = join(input_path, 'train-labels.idx1-ubyte')
    test_images_filepath = join(input_path, 't10k-images.idx3-ubyte')
    test_labels_filepath = join(input_path, 't10k-labels.idx1-ubyte')


    # Load MINST dataset
    mnist_dataloader = MnistDataloader(training_images_filepath, training_labels_filepath, test_images_filepath, test_labels_filepath)
    (x_train, y_train), (x_test, y_test) = mnist_dataloader.load_data()


    # Example of reconstructing a digit
    digit_idx = np.random.randint(0, len(x_test))
    digit_image = x_test[digit_idx]
```

```python

    K = [100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]


    # Prepare the figure for subplots
    fig, axs = plt.subplots(2, 10, figsize=(18, 4))  # 1 row, columns for each K plus one for the original

    # Display the original digit in the first column
    axs[0, 0].imshow(digit_image, cmap='gray')
    axs[0, 0].set_title('Original Number {}'.format(y_train[digit_idx]))
    axs[0, 0].axis('off')



    for i_fig, n_clusters in enumerate(K):
        # Load the model
        try:
            model = joblib.load(f"K-means/Result/Model/{n_clusters}-clusters-model.joblib")
        except FileNotFoundError:
            print(f"cd to folder EEC289A, run run_kmeans.py first.")
            exit()

        # Reconstruct the digit
        reconstructed_image = reconstruct_digit(digit_image, model)

        # Display reconstructed digit for this K
        if i_fig < 9:
            axs[0, i_fig + 1].imshow(reconstructed_image, cmap='gray')
            axs[0, i_fig + 1].set_title(f'K={n_clusters}')
            axs[0, i_fig + 1].axis('off')
        else:
            axs[1, i_fig - 9].imshow(reconstructed_image, cmap='gray')
            axs[1, i_fig - 9].set_title(f'K={n_clusters}')
            axs[1, i_fig - 9].axis('off')

    # plt.tight_layout()
    plt.savefig(f"K-means/Result/Digits/reconstruct-test-digit.png", dpi=300)
```