CpE 520: HW #4

West Virginia University

Ali Zafari

Contents

1 *MNIST* Dataset

1.1 Installing *MNSIT* Package

At first to load the MNIST dataset, we install the mnist package of python:

```
[1]: !pip install mnist
```

```
Collecting mnist
  Downloading https://files.pythonhosted.org/packages/c6/c4/5db3bfe009f8d71f1d53
2bbadbd0ec203764bba3a469e4703a889db8e5e0/mnist-0.2.2-py2.py3-none-any.whl
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages
(from mnist) (1.18.2)
Installing collected packages: mnist
Successfully installed mnist-0.2.2
```

1.2 Loading the *MNIST* dataset

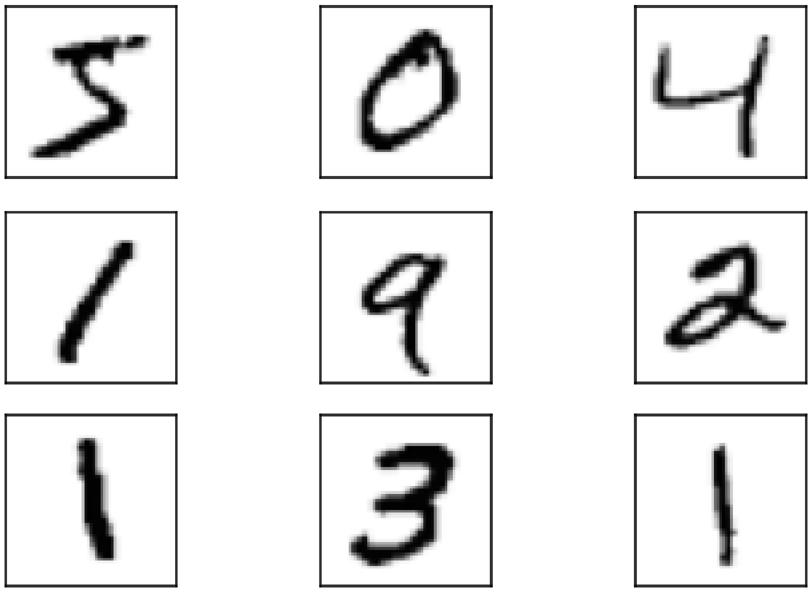
The code below will import the necessary libraries and loads the training part of the MNIST dataset which will be in a matrix shape of (60000, 28, 28)

```
[ ]: import mnist
import os
import numpy as np
import matplotlib.pyplot as plt

mnist.temporary_dir = lambda: os.getcwd()

# Each of these functions first downloads the data and returns a numpy array.
train_images = mnist.train_images()

assert train_images.shape == (60000, 28, 28), 'train_imgages shape is not␣
 ↪correct'
```

1.3 Showing a Sample of Dataset

This code snippet will prints the first 9 images as an example:

```
[3]: fig, ax = plt.subplots(3, 3)
for i in range(3):
    for j in range(3):
        ax[i][j].imshow(train_images[3 * i + j], cmap='Greys')
        ax[i][j].xaxis.set_visible(False)
        ax[i][j].yaxis.set_visible(False)
```



1.4 Reshaping the Training Images

To do matrix operations on this data we reshape each image of 28x28 to a vector of 784 element, so the new shape of the data will be like (60000, 784).

```
[4]: train_images_reshaped = train_images.reshape((train_images.shape[0], 28*28))
     print(f'Train images before reshape: {train_images.shape}')
     print(f'Train images after  reshape: {train_images_reshaped.shape}')

     assert all(train_images[0, 20] == train_images_reshaped[0, (20)*28:(20+1)*28]),␣
      ↪"train reshaping is wrong"
```

```
Train images before reshape: (60000, 28, 28)
Train images after  reshape: (60000, 784)
```

2 Implementing K-means Algorithm in Functions

All the functions that implement the K-means algorithm are listed below, it is sufficient to call the function ***kmeans_algorithm_with_distortion*** and it will use the other functions. The output of this function is the final centroids and final cluster number that is assigned to each image.

[ ]:
```
def initialize_centroids(X, K):
    return X[np.random.choice(X.shape[0], K, replace=False), :]



def find_closest_centroids(X, centroids):
    #print('find_closest_centroids_started')
    K = centroids.shape[0]
    idx = np.zeros(X.shape[0])

    for n in range(X.shape[0]):
        min_distance = np.inf
        j = 0
        for m in range(K):

            norm = np.linalg.norm(centroids[m, :]-X[n, :])
            #print(m, n, norm)
            if norm < min_distance:
                min_distance = norm
                j = m
        idx[n] = j
    #print('find_closest_centroids_finished')
    return idx


def find_closest_centroids_numpyed(X, centroids): #maybe faster! but is not␣
 ↪memory friendly
    #print('find_closest_centroids_numpyed_started')
    K = centroids.shape[0]

    X_minus_Cs = np.array([X-centroids[i, :] for i in range(K)])
    norms = np.linalg.norm(X_minus_Cs, axis=2).T
    idx = np.argmin(norms, axis=1)
    #print('find_closest_centroids_numpyed_finished')
    return idx



def compute_centroids(X, idx, K, prev_centroids):
    centroids = np.copy(prev_centroids)

    for i in np.unique(idx).astype(int):
        centroids[i, :] = np.mean(X[idx.astype(int) == i, :], axis=0)
    return centroids


def show_centroids(centroids, K):
    fig, ax = plt.subplots(K//5, 5)
    ax = ax.reshape(K//5, 5)
    for i in range(K//5):
        for j in range(5):
            ax[i][j].imshow(centroids[5 * i + j].reshape(28, 28), cmap='Greys')
            ax[i][j].xaxis.set_visible(False)
            ax[i][j].yaxis.set_visible(False)
    plt.show()


def distortion(final_centroids, final_idx):
    J = 0

    for i in np.unique(final_idx).astype(int):
        X_desired = X[final_idx.astype(int) == i, :]
        X_desired = X_desired-final_centroids[i, :]
        J += np.mean(X_desired**2)
    return J


def kmeans_algorithm(X, initial_centroids, iner_max_iters, K):
    prev_centroids = initial_centroids

    for i in range(iner_max_iters):
        idx = find_closest_centroids(X, prev_centroids)
        new_centroids = compute_centroids(X, idx, K, prev_centroids)

        centroids_change_metric = np.linalg.norm(new_centroids - prev_centroids)
        print(f'{i+1}th iteration: centroids got changed as␣
 ↪{centroids_change_metric}.')
        print(f'inner J is: {distortion(new_centroids, idx)}')
        if centroids_change_metric < 0.01:
            print(f'K-Means algorithm converged on {i}(th) iteration.')
            break
        prev_centroids = new_centroids
        print('------------------------------------------')
    return new_centroids, idx


def kmeans_algorithm_with_distortion(X, max_iters, iner_max_iters, K):
    lowest_J = np.inf
    new_centroids = np.zeros((K, X.shape[1]))
    new_idx = np.zeros(X.shape[0])

    for i in range(max_iters):
        print(f'iteration {i+1} of {max_iters} is going on...')
        new_initial_centroids = initialize_centroids(X, K)
        new_centroids, new_idx = kmeans_algorithm(X, new_initial_centroids,␣
↪iner_max_iters, K)
        new_J = distortion(new_centroids, new_idx)
        print(f'J to compare is: {new_J}')
        if new_J < lowest_J:
            lowest_J = new_J
            final_centroids = new_centroids
            final_idx = new_idx

    return final_centroids, final_idx
```

3 Using the K-means to Answer Questions

3.1 Parameters to Pass to the Algorithm

There are three parameter that must be set for the algorithm:

- **K**: Number of Clusters
- **max_iter**: the number of iterations, it sets the number that K-means algorithm will be ran (every time with a randomly chosen centroids) and the one with the lowest distortion error will be picked up as the final solution
- **iner_max_iters**: the number of inner iterations, it sets the number that K-means algorithm tries to enhance the centroids until the centroids do not move much (with a fixed initial centroids)

Obviously, the whole dataset will be fed into our code, and we will deivide all its entries by 255 to make them less than one.

```
[ ]: K = 20
     max_iters = 10
     iner_max_iters = 70

     X = train_images_reshaped/255
```

3.2 Getting Final Centroids and Clusters

The main function call to get the final centroids and final clusters to which the images are assigned. This function is called three times and after that the centroids will be showed as images.

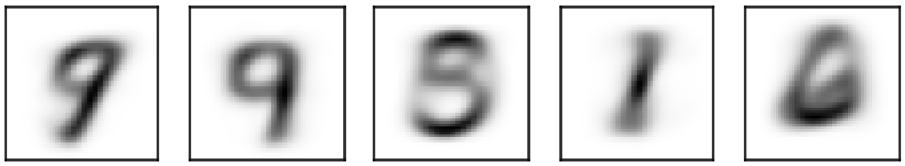
- **K=5**
- **K=10**
- **K=20**

```
[ ]: final_centroids, final_idx = kmeans_algorithm_with_distortion(X, max_iters,␣
      ↪iner_max_iters, K)
```

3.3 5 Centroids: Images and Clusters

Centroids are shown as 28x28 images below from left to right with indixes 0 to 5:

```
[ ]: # K=5, max_iters=15, max_inner_iters=40
     show_centroids(final_centroids, K)
```



Number of images assigned to each centroids, namely clusters:

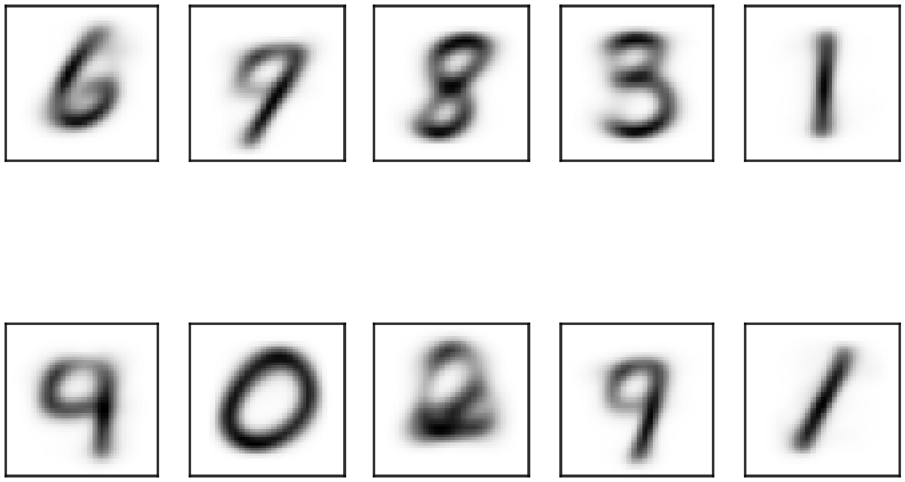
```
[ ]: for i in np.unique(final_idx).astype(int):
         print(f'number of input in idx={i} cluster: ', X[final_idx.astype(int) == i,␣
      ↪:].shape[0])
```

```
number of input in idx=0 cluster:  11562
number of input in idx=1 cluster:  10252
number of input in idx=2 cluster:  13071
number of input in idx=3 cluster:  13433
number of input in idx=4 cluster:  11682
```

3.4 10 Centroids: Images and Clusters

Centroids are shown as 28x28 images below from left to right with indixes 0 to 10:

```
[ ]: # K=10, max_iters=10, max_inner_iters=50
     show_centroids(final_centroids, K)
```



Number of images assigned to each centroids, namely clusters:

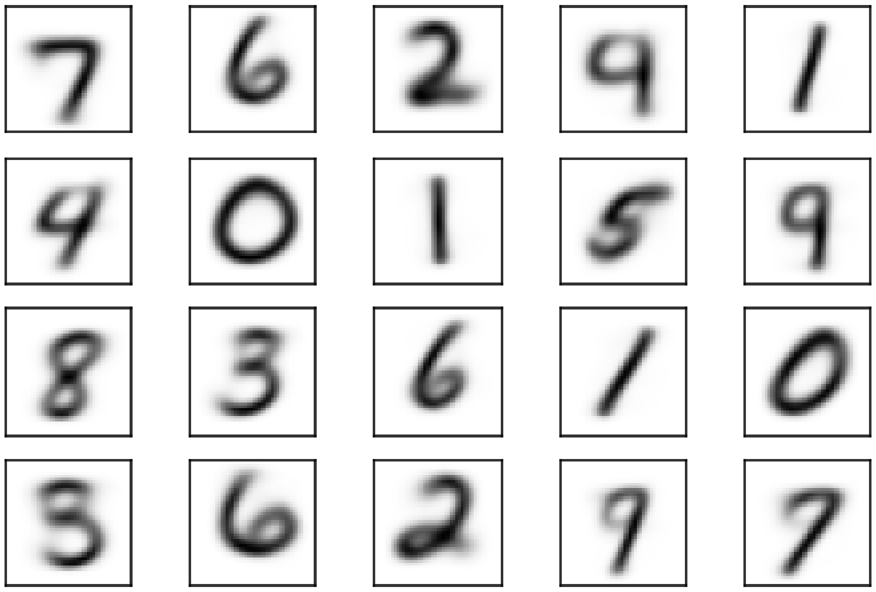
```
[ ]: for i in np.unique(final_idx).astype(int):
         print(f'number of input in idx={i} cluster: ', X[final_idx.astype(int) == i,␣
      ↪:].shape[0])
```

```
number of input in idx=0 cluster:  6059
number of input in idx=1 cluster:  6459
number of input in idx=2 cluster:  6159
number of input in idx=3 cluster:  7621
number of input in idx=4 cluster:  5298
number of input in idx=5 cluster:  5196
number of input in idx=6 cluster:  4705
number of input in idx=7 cluster:  6660
number of input in idx=8 cluster:  6941
number of input in idx=9 cluster:  4902
```

3.5 20 Centroids: Images and Clusters

Centroids are shown as 28x28 images below from left to right with indixes 0 to 5:

[8]:
```
# K=20, max_iters=10, max_inner_iters=70
show_centroids(final_centroids, K)
```



Number of images assigned to each centroids, namely clusters:

[9]:
```
for i in np.unique(final_idx).astype(int):
    print(f'number of input in idx={i} cluster: ', X[final_idx.astype(int) == i,␣
 ↪:].shape[0])
```

```
number of input in idx=0 cluster:  2265
number of input in idx=1 cluster:  2507
number of input in idx=2 cluster:  2685
number of input in idx=3 cluster:  2916
number of input in idx=4 cluster:  2793
number of input in idx=5 cluster:  3026
number of input in idx=6 cluster:  2622
number of input in idx=7 cluster:  3083
number of input in idx=8 cluster:  3082
number of input in idx=9 cluster:  4255
number of input in idx=10 cluster:  4332
number of input in idx=11 cluster:  4250
number of input in idx=12 cluster:  2651
number of input in idx=13 cluster:  2156
number of input in idx=14 cluster:  2637
number of input in idx=15 cluster:  4694
number of input in idx=16 cluster:  1514
number of input in idx=17 cluster:  2454
number of input in idx=18 cluster:  3369
number of input in idx=19 cluster:  2709
```

4 More Comments and Comparison with Global PCA

- **K=5**

  It can be seen from section **3.3** that the number of clusters is limited and the centroids are like each other. They result in *5* approximately equal distributed clusters.

- **K=10**

  As we can see from section **3.4**, when the number of centroids is fixed on 10, they are very similar to the original digits *0* to *9* although there is two centroids that are like digit *1* and unfortunately it has not predicted digit *5*. As it can be seen from their images, we could say that omitting the digit *5* is the result of its resemblance to digit *2*. In the previous homework (HW3), when we did a global PCA on this same dataset, we printed the first 20 eigenvectors as images. Between them there were also some digits visible such as digits 0, 7, 3, 6, 8 but they were not very clear in comparison to the centroids we have found here. This conclusion seems to be reasonable because the centroids are actually the mean over the same digits, so when we plot them, they will look like actual digits. But the eigenvectors are not like this. Eignevectors try to keep the part of matrix in which most of the information exists, and there is no reason for them to be like an actual image of a digit.

- **K=20**

  When the number of centroids was increased to 20 as it is in section **3.5**, we have more free space to detect different digits. So in 20 centroids we could see all the digits from *0* to *9* and obviously some of them will be repeated.