ASHUTOSH KUMAR SINGH
19CS30008

Question 9

Values of N and n :
We choose 100 images of each digit as our training set.
So, N = no. of training samples = 100 * 10 = 1000
and, n = dimension of feature vector after flattening = 28 * 28 = 784

The same result is also evident from the code output:

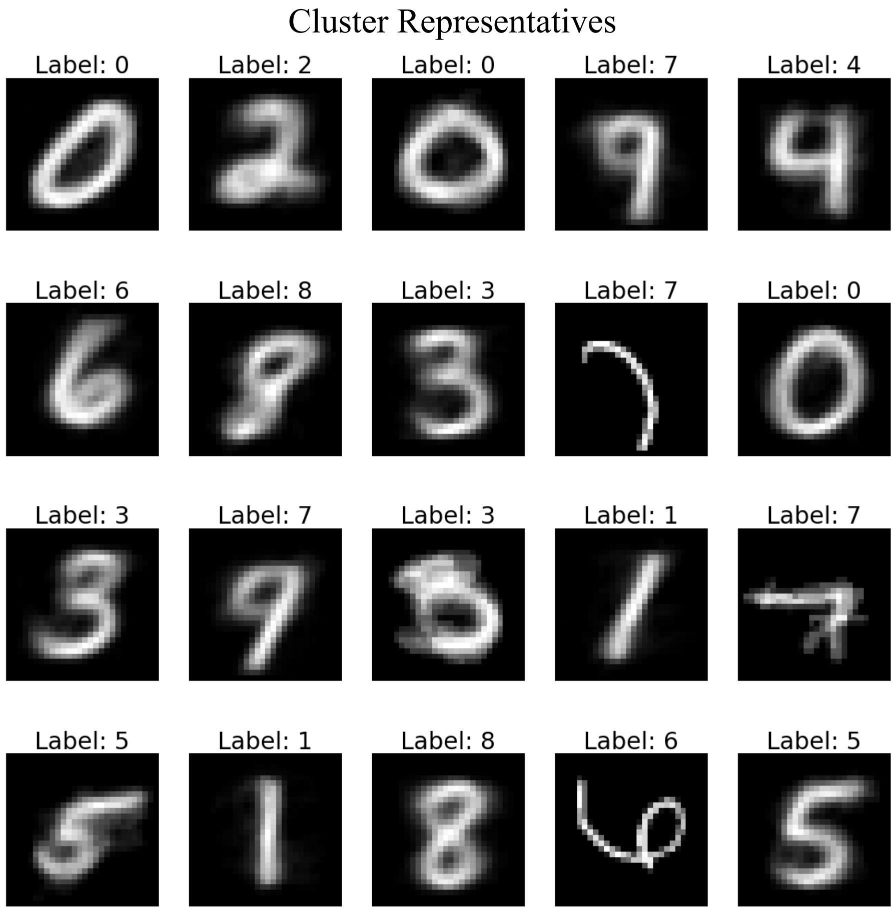
```
x_train: (1000, 784), y_train: (1000,), x_test: (50, 784), y_test:(50,)
Number of training samples, N = 1000
Dimension of feature vector, n = 784
```

Convergence Criteria :
We say that the K-Means algorithm has converged if the change in the cluster representative vectors in two successive iterations is lower than a specific threshold. We can quantify the change in the cluster representative vectors using the norm of the difference vector (the difference vector is basically previous cluster representatives - new cluster representatives), and the threshold considered in this case is 10^{-6}.

```
converged = True
for curr_cluster in range(self.num_clusters):
    if np.linalg.norm(self.prev_centers[curr_cluster] - self.centers[curr_cluster], ord=2) > CONVERGENCE_LIMIT:
        converged = False
```

(i) Random initialization of the cluster representatives

(a)



The number of iterations required to converge is **49** in this case.
Loss (J_clust) after 49 iterations = **35.33122**

In our calculations, we have considered the function J_clust as:

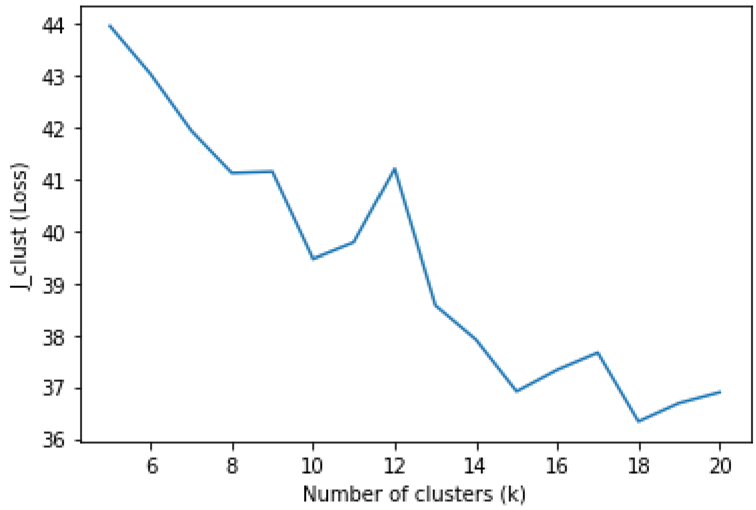
$$J_{clust} = \frac{\|x_1 - z_{c_1}\|^2 + \|x_2 - z_{c_2}\|^2 + \ldots + \|x_n - z_{c_n}\|^2}{n}$$

(b) Test Accuracy = **0.58 (= 58%)** (after choosing 50 test images)

```
Converged after iteration: 49
J_clust: 35.33122978860102
Test Accuracy: 0.58
```

(c)

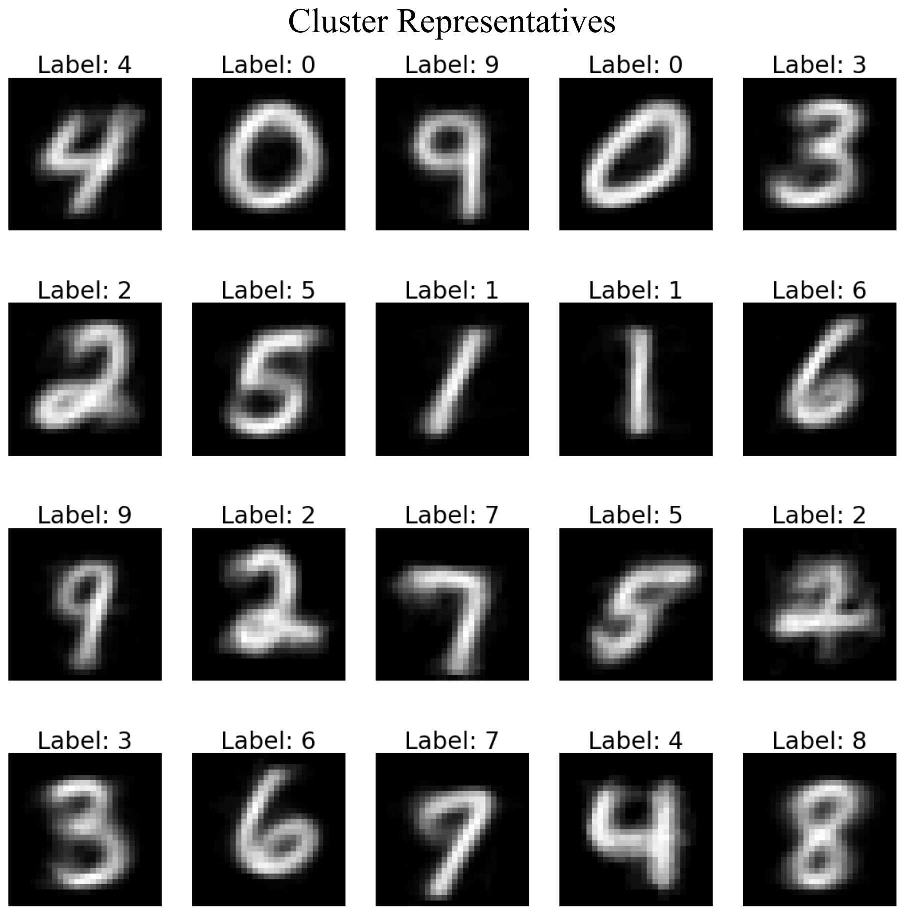
k	J^{clust}
5	43.96431
6	43.03560
7	41.94476
8	41.12340
9	41.149697
10	39.468275
11	39.784229
12	41.205235
13	38.575877
14	37.913875
15	36.912872
16	37.329615
17	37.66097
18	36.33203
19	36.68637
20	36.89452



From the graph of J_clust v/s Number of clusters, we can see that the optimal value of k comes to be **18**.

(ii) Choosing initial cluster representatives from the given data set

(a)



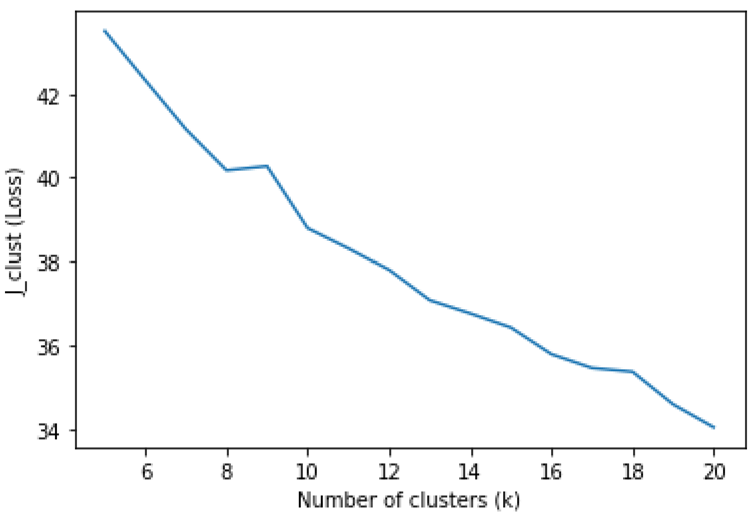
The number of iterations required to converge is **26** in this case.
Loss (J_clust) after 26 iterations = **33.94842**

(b) Test Accuracy = **0.74 (= 74%)** (after choosing 50 test images)

```
Converged after iteration: 26
J_clust: 33.94842967207276
Test Accuracy: 0.74
```

(c)

k	J^{clust}
5	43.50424
6	42.32438
7	41.15703
8	40.17545
9	40.27544
10	38.79539
11	38.31619
12	37.80257
13	37.08120
14	36.76557
15	36.42989
16	35.79222
17	35.46301
18	35.37364
19	34.594038
20	34.046516



From the graph of J_clust v/s Number of clusters, we can see that the optimal value of k comes to be **20**.

Note that the optimal number of clusters is greater than 10 (the number of digits), because different people have different ways of writing the same digit.

Yes, the choice of the initial condition affects the performance of the k-clustering algorithm. The method of choosing initial cluster representatives from the given data set is better than random assignment. In case of random assignment, the test accuracy is mostly lower as compared to choosing from the data set. Also, the accuracy in random assignment over a test set varies greatly over multiple runs of the K-Means algorithm, proving that the results obtained are not very stable. Also, in case of random assignment, the algorithm converges to several local minima, thus giving poor results. We also see that the images of the cluster representatives sometimes do not show any digit during random assignment because here it may happen that we have initialized the cluster representative to something very distant from the training data set. But all these problems are solved when choosing initial cluster representatives from the given data set.

The code for this problem is attached below.

[7]:
```
import random
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import mnist

MAX_NUM_ITERATIONS = 100
CONVERGENCE_LIMIT = 1e-6
```

[8]:
```
class KMeans():
    # initialize the KMeans object
    def __init__(self, x_train, y_train, num_clusters=3, init_type='choose'):
        self.data = x_train
        self.targets = y_train
        self.num_clusters = num_clusters
        self.sample_size = x_train.shape[0]
        self.feature_size = x_train.shape[1]

        if init_type == 'choose':
            self.centers = np.copy(self.data[np.random.choice(
                self.sample_size, self.num_clusters, replace=(
                    False if self.num_clusters <= self.sample_size else True))])
        else:       # init_type == 'random'
            self.centers = np.random.uniform(
                size=(self.num_clusters, self.feature_size))

        self.prev_centers = np.copy(self.centers)
        self.cluster_labels = np.zeros(self.sample_size, dtype=int)

    # function to get the norm of 2 vectorized feature vectors
    def diff_norm(self, p, q):
        return np.linalg.norm(p - q, ord=2, axis=1)

    # function to assign clusters to data points based on minimum norm
    def assign_clusters(self):
        for i in range(self.sample_size):
            norms = self.diff_norm(self.data[i], self.centers)
            self.cluster_labels[i] = np.argmin(norms)

    # function to update the centers (cluster representatives)
    def update_centers(self):
        self.prev_centers = np.copy(self.centers)
        for curr_cluster in range(self.num_clusters):
            curr_group = self.data[self.cluster_labels == curr_cluster]
            if len(curr_group) != 0:
                self.centers[curr_cluster] = np.mean(curr_group, axis = 0)
            else:
                self.centers[curr_cluster] = np.zeros(self.feature_size)
```

```
    # function to calculate the J_clust value
    def calculate_loss(self):
        return np.mean(np.square(self.diff_norm(
            self.data, self.centers[self.cluster_labels])))

    # function to train the K-Means algorithm
    def train(self, details=True):
        for i in range(MAX_NUM_ITERATIONS):
            self.assign_clusters()
            self.update_centers()
            loss = self.calculate_loss()
            if details:
                print("Iteration {} Loss: {}".format(i + 1, loss))
                print("----------------------------")
            converged = True
            for curr_cluster in range(self.num_clusters):
                if np.linalg.norm(self.prev_centers[curr_cluster] -
                        self.centers[curr_cluster], ord=2) > CONVERGENCE_LIMIT:
                    converged = False
            if converged:
                print("k = {} Loss: {}".format(self.num_clusters, loss))
                # print("Converged after iteration: {}".format(i + 1))
                # print("J_clust: {}".format(loss))
                break

    # function to get labels for the cluster representatives
    def get_center_labels(self):
        center_labels = np.zeros(self.num_clusters)
        for i in range(self.num_clusters):
            count = np.bincount(self.targets[self.cluster_labels == i])
            if len(count) > 0:
                center_labels[i] = np.argmax(count)
        return center_labels

    # function to predict labels for new test examples
    def predict(self, test_data):
        labels = np.zeros(test_data.shape[0], dtype=int)
        for i in range(test_data.shape[0]):
            labels[i] = np.argmin(self.diff_norm(test_data[i], self.centers))
        center_labels = self.get_center_labels()
        return center_labels[labels]
```

[ ]:
```
# function to load the MNIST data in the required format
def load_mnist_data():
    (x_train, y_train), (x_test, y_test) = mnist.load_data()
    x_train = x_train.reshape(x_train.shape[0], -1) / 255.0
```

```python
    x_test = x_test.reshape(x_test.shape[0], -1) / 255.0

    digits = []
    targets = []
    for i in range(10):
        images = x_train[y_train == i]
        digits.append(images[np.random.choice(
            len(images), 100, replace=False)])
        targets.append(np.full((100,), i))

    x_train = np.vstack(digits)
    y_train = np.hstack(targets)

    order = np.random.permutation(x_train.shape[0])
    x_train = x_train[order]
    y_train = y_train[order]

    ind = np.random.choice(x_test.shape[0], 50)
    x_test = x_test[ind]
    y_test = y_test[ind]
    return (x_train, y_train), (x_test, y_test)

# function to plot the J_clust value varying the number of clusters
def plot_J():
    k = np.arange(start=5, stop=21, step=1, dtype=int)
    (x_train, y_train), (x_test, y_test) = load_mnist_data()
    J = []
    for i in k:
        kmeans = KMeans(x_train, y_train, i, 'choose')
        kmeans.train(details=False)
        J.append(kmeans.calculate_loss())

    plt.plot(k, J)
    plt.xlabel("Number of clusters (k)")
    plt.ylabel("J_clust (Loss)")
    plt.show()

# function to plot the cluster representatives
def plot_centers(kmeans):
    center_images = np.copy(kmeans.centers.reshape(
        kmeans.num_clusters, 28, 28)) * 255
    center_labels = kmeans.get_center_labels()

    plot = plt.figure(figsize=(20, 20))
    rows = 4
    cols = 5
    for i in range(kmeans.num_clusters):
```

```python
        plot.add_subplot(rows, cols, i + 1)
        plt.imshow(center_images[i], cmap='gray')
        plt.title(f"Label: {int(center_labels[i])}", fontsize=30)
        plt.axis('off')
    plt.show()

# main function to perform all required tasks
def main():
    random.seed(40)
    np.random.seed(40)
    (x_train, y_train), (x_test, y_test) = load_mnist_data()
    print("x_train: {}, y_train: {}, x_test: {}, y_test:{}".format(
        x_train.shape, y_train.shape, x_test.shape, y_test.shape))
    print("Number of training samples, N = {}".format(x_train.shape[0]))
    print("Dimension of feature vector, n = {}".format(x_train.shape[1]))
    kmeans = KMeans(x_train, y_train, 20, 'choose')
    kmeans.train(details=True)
    predictions = kmeans.predict(x_test)
    print("Test Accuracy: {}".format(np.mean(predictions == y_test)))
    print()

    plot_centers(kmeans)
    plot_J()

if __name__ == '__main__':
    main()
```