Sardar Patel Institute of Technology, Mumbai
Department of Electronics and Telecommunication Engineering
B.E. Sem-VII (2022-2023)
OEIT6 - Data Analytics

Experiment: Apriori Algorithm

Name: Pushkar Sutar **Roll No. 2019110060**

Objective : Apply K means algorithm to MNIST dataset and few other datasets.

Code and Output:

Kmeans.py

```
from math import *
import random
from copy import deepcopy
import numpy as np


def argmin(values):
    return min(enumerate(values), key=lambda x: x[1])[0]


def avg(values):
    return float(sum(values)) / len(values)


def readfile(filename):
    '''
    File format: Each line contains a comma separated list of real numbers,
representing a single point.
    Returns a list of N points, where each point is a d-tuple.
    '''
    data = []
    with open(filename, 'r') as f:
        data = f.readlines()
    data = [tuple(map(float, line.split(','))) for line in data]
    return data


def writefile(filename, means):
    '''
    means: list of tuples
    Writes the means, one per line, into the file.
    '''
    if filename == None: return
```

```
    with open(filename, 'w') as f:
        for m in means:
            f.write(','.join(map(str, m)) + '\n')
    print('Written means to file ' + filename)


def distance_euclidean(p1, p2):
    '''
    p1: tuple: 1st point
    p2: tuple: 2nd point

    Returns the Euclidean distance b/w the two points.
    '''

    distance = None

    # TODO [task1]:
    # Your function must work for all sized tuples.
    dist = [(x1 - x2) ** 2 for x1, x2 in zip(p1, p2)]
    distance = sqrt(sum(dist))
    ########################################
    return distance


def distance_manhattan(p1, p2):
    '''
    p1: tuple: 1st point
    p2: tuple: 2nd point

    Returns the Manhattan distance b/w the two points.
    '''

    # k-means uses the Euclidean distance.
    # Changing the distant metric leads to variants which can be more/less
robust to outliers,
    # and have different cluster densities. Doing this however, can sometimes
lead to divergence!

    distance = None

    # TODO [task1]:
    # Your function must work for all sized tuples.
    distance = sum([abs(x1 - x2) for x1, x2 in zip(p1, p2)])
    ########################################
    return distance


def initialization_forgy(data, k):
    '''
```

```
    data: list of tuples: the list of data points
    k: int: the number of cluster means to return

    Returns a list of tuples, representing the cluster means
    '''

    means = []
    means = random.sample(data, k)

    # TODO [task1]:
    # Use the Forgy algorithm to initialize k cluster means.

    ########################################
    assert len(means) == k
    return means


def initialization_kmeansplusplus(data, distance, k):
    '''
    data: list of tuples: the list of data points
    distance: callable: a function implementing the distance metric to use
    k: int: the number of cluster means to return
    Returns a list of tuples, representing the cluster means
    '''

    means = []
    # Initialising the first mean randomly from the data
    rand_index = random.randint(0, len(data) - 1)
    rand_point = data[rand_index]
    means.append(rand_point)

    num_done = 1  # The number of mean points so far
    # Iterating till we have k mean points
    while num_done < k:
        sum_tot = 0  # The sum of the distances so far
        point_distances = dict()  # The square of the distance of a point to the
nearest mean is stored in a dictionary
        for data_point in data:
            # Initialising the nearest mean and the nearest distance for each point
            nearest_mean = means[0]
            nearest_dist = distance(data_point, means[0])

            # This is done separately because means[1] won't be valid if only one
mean has been added so far
            if len(means) == 1:
                point_distances[data_point] = nearest_dist ** 2  # Storing the
square of the nearest distance
                sum_tot += nearest_dist ** 2  # Updating the sum so far
                continue
```

```
        # Going through all the means to determine the nearest mean for the
data point
        for mean_point in means[1:]:
            dist = distance(data_point, mean_point)
            if dist < nearest_dist:  # Updating the nearest mean and the nearest
distance
                nearest_dist = dist
                nearest_mean = mean_point

        point_distances[data_point] = nearest_dist ** 2  # Storing the square
of the nearest distance
        sum_tot += nearest_dist ** 2  # Updating the sum so far

    for data_point in data:  # Normalizing the distribution
        point_distances[data_point] /= sum_tot * 1.0

    # This is basically sampling from a multinomial distribution
    rand_num = random.random()  # Generating the random number
    for data_point in data:  # Checking if the random number lies in a
        # particular interval and if it does, we set the data point
corresponding to that interval as our mean and we break off
        # we also check if the probability of that data point is not zero
        # (it can be zero only if it has been selected before, so this ensures
that the same mean is not selected twice)
        if rand_num < point_distances[data_point] and
point_distances[data_point] > 0:
            means.append(data_point)
            break
        rand_num -= point_distances[
            data_point]  # This step is needed as this interval is done and we
need to update the value for the next interval
    num_done += 1
  # TODO [task3]:
  # Use the kmeans++ algorithm to initialize k cluster means.
  # Make sure you use the distance function given as parameter.

  # NOTE: Provide extensive comments with your code.

  ########################################
  assert len(means) == k
  return means


def iteration_one(data, means, distance):
    '''
    data: list of tuples: the list of data points
    means: list of tuples: the current cluster centers
    distance: callable: function implementing the distance metric to use
```

```
    Returns a list of tuples, representing the new cluster means after 1
iteration of k-means clustering algorithm.
    '''

    new_means = []
    k = len(means)
    dimension = len(data[0])

    # TODO [task1]:
    # You must find the new cluster means.
    # Perform just 1 iteration (assignment+updation)
    new_means = [tuple(0 for i in range(dimension))] * k
    counts = [0.0] * k
    for point in data:
        closest = 0
        min_dist = float('Inf')
        for i in range(k):
            d = distance(point, means[i])
            if d < min_dist:
                min_dist = d
                closest = i
        new_means[closest] = tuple([sum(x) for x in zip(new_means[closest],
point)])
        counts[closest] += 1

    for i in range(k):
        # import pdb; pdb.set_trace()
        if counts[i] == 0:
            new_means[i] = means[i]
        else:
            new_means[i] = tuple(t / counts[i] for t in new_means[i])
    ########################################
    return new_means


def hasconverged(old_means, new_means, epsilon=1e-1):
    '''
    old_means: list of tuples: The cluster means found by the previous
iteration
    new_means: list of tuples: The cluster means found by the current iteration

    Returns true iff no cluster center moved more than epsilon distance.
    '''

    converged = False

    # TODO [task1]:
    # Use Euclidean distance to measure centroid displacements.
    for i in range(len(old_means)):
```

```
        p = [abs(x1 - x2) > epsilon for x1, x2 in zip(old_means[i],
new_means[i])]
        if True in p:
            return False
    converged = True
    ########################################
    return converged


def iteration_many(data, means, distance, maxiter, epsilon=1e-1):
    '''
    maxiter: int: Number of iterations to perform

    Uses the iteration_one function.
    Performs maxiter iterations of the k-means clustering algorithm, and saves
the cluster means of all iterations.
    Stops if convergence is reached earlier.

    Returns:
    all_means: list of (list of tuples): Each element of all_means is a list of
the cluster means found by that iteration.
    '''

    all_means = []
    all_means.append(means)

    # TODO [task1]:
    # Make sure you've implemented the iteration_one, hasconverged functions.
    # Perform iterations by calling the iteration_one function multiple times.
    # Stop only if convergence is reached, or if max iterations have been
exhausted.
    # Save the results of each iteration in all_means.
    # Tip: use deepcopy() if you run into weirdness.
    means_copy = deepcopy(means)
    for i in range(maxiter):
        new_means = iteration_one(data, means_copy, distance)
        all_means.append(new_means)
        if hasconverged(means_copy, new_means, epsilon):
            break
        means_copy = new_means
    ########################################

    return all_means


def performance_SSE(data, means, distance):
    '''
    data: list of tuples: the list of data points
    means: list of tuples: representing the cluster means
```

```
    Returns: The Sum Squared Error of the clustering represented by means, on
the data.
    '''

    sse = 0

    # TODO [task1]:
    # Calculate the Sum Squared Error of the clustering represented by means, on
the data.
    # Make sure to use the distance metric provided.
    for point in data:
        min_dist = float('Inf')
        for i in range(len(means)):
            d = distance(point, means[i])
            if d < min_dist:
                min_dist = d
        sse += min_dist * min_dist
    ########################################
    return sse


#############################################################################
##                        DO NOT EDIT THE FOLLWOING                        ##
#############################################################################


import sys
import argparse
import matplotlib.pyplot as plt
from itertools import cycle
from pprint import pprint as pprint


def parse():
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input', dest='input', type=str, help='Required.
Dataset filename')
    parser.add_argument('-o', '--output', dest='output', type=str, help='Output
filename')
    parser.add_argument('-iter', '--iter', '--maxiter', dest='maxiter',
type=int, default=10000,
                        help='Maximum number of iterations of the k-means
algorithm to perform. (may stop earlier if convergence is achieved)')
    parser.add_argument('-e', '--eps', '--epsilon', dest='epsilon', type=float,
default=1e-1,
                        help='Minimum distance the cluster centroids move b/w
two consecutive iterations for the algorithm to continue.')
```

```
    parser.add_argument('-init', '--init', '--initialization', dest='init',
type=str, default='forgy',
                        help='The initialization algorithm to be used. {forgy,
randompartition, kmeans++}')
    parser.add_argument('-dist', '--dist', '--distance', dest='dist', type=str,
default='euclidean',
                        help='The distance metric to be used. {euclidean,
manhattan}')
    parser.add_argument('-k', '--k', dest='k', type=int, default=5, help='The
number of clusters to use.')
    parser.add_argument('-verbose', '--verbose', dest='verbose', type=bool,
default=False, help='Turn on/off verbose.')
    parser.add_argument('-seed', '--seed', dest='seed', type=int, default=0,
help='The RNG seed.')
    parser.add_argument('-numexperiments', '--numexperiments',
dest='numexperiments', type=int, default=1,
                        help='The number of experiments to run.')
    _a = parser.parse_args()

    if _a.input is None:
        print
        'Input filename required.\n'
        parser.print_help()
        sys.exit(1)

    args = {}
    for a in vars(_a):
        args[a] = getattr(_a, a)

    if _a.init.lower() in ['random', 'randompartition']:
        args['init'] = initialization_randompartition
    elif _a.init.lower() in ['k++', 'kplusplus', 'kmeans++', 'kmeans',
'kmeansplusplus']:
        args['init'] = initialization_kmeansplusplus
    elif _a.init.lower() in ['forgy', 'frogy']:
        args['init'] = initialization_forgy
    else:
        print
        'Unavailable initialization function.\n'
        parser.print_help()
        sys.exit(1)

    if _a.dist.lower() in ['manhattan', 'l1', 'median']:
        args['dist'] = distance_manhattan
    elif _a.dist.lower() in ['euclidean', 'euclid', 'l2']:
        args['dist'] = distance_euclidean
    else:
        print
        'Unavailable distance metric.\n'
```

```
        parser.print_help()
        sys.exit(1)

    print
    '-' * 40 + '\n'
    print
    'Arguments:'
    pprint(args)
    print
    '-' * 40 + '\n'
    return args


def visualize_data(data, all_means, args):
    print
    'Visualizing...'
    means = all_means[-1]
    k = args['k']
    distance = args['dist']
    clusters = [[] for _ in range(k)]
    for point in data:
        dlist = [distance(point, center) for center in means]
        clusters[argmin(dlist)].append(point)

    # plot each point of each cluster
    colors = cycle('rgbwkcmy')

    for c, points in zip(colors, clusters):
        x = [p[0] for p in points]
        y = [p[1] for p in points]
        plt.scatter(x, y, c=c)

    # plot each cluster centroid
    colors = cycle('krrkgkgr')
    colors = cycle('rgbkkcmy')

    for c, clusterindex in zip(colors, range(k)):
        x = [iteration[clusterindex][0] for iteration in all_means]
        y = [iteration[clusterindex][1] for iteration in all_means]
        plt.plot(x, y, '-x', c=c, linewidth='1', mew=15, ms=2)
    plt.axis('equal')
    plt.show()


def visualize_performance(data, all_means, distance):
    errors = [performance_SSE(data, means, distance) for means in all_means]
    plt.plot(range(len(all_means)), errors)
    plt.title('Performance plot')
    plt.xlabel('Iteration')
```

```
    plt.ylabel('Sum Squared Error')
    plt.show()


if __name__ == '__main__':

    args = parse()
    # Read data
    data = readfile(args['input'])
    print
    'Number of points in input data: {}\n'.format(len(data))
    verbose = args['verbose']

    totalSSE = 0
    totaliter = 0

    for experiment in range(args['numexperiments']):
        print
        'Experiment: {}'.format(experiment + 1)
        random.seed(args['seed'] + experiment)
        print
        'Seed: {}'.format(args['seed'] + experiment)

        # Initialize means
        means = []
        if args['init'] == initialization_forgy:
            means = args['init'](data, args['k'])  # Forgy doesn't need distance
metric
        else:
            means = args['init'](data, args['dist'], args['k'])

        if verbose:
            print
            'Means initialized to:'
            print
            means
            print
            ''

        # Run k-means clustering
        all_means = iteration_many(data, means, args['dist'], args['maxiter'],
args['epsilon'])

        SSE = performance_SSE(data, all_means[-1], args['dist'])
        totalSSE += SSE
        totaliter += len(all_means) - 1
        print
        'Sum Squared Error: {}'.format(SSE)
        print
```

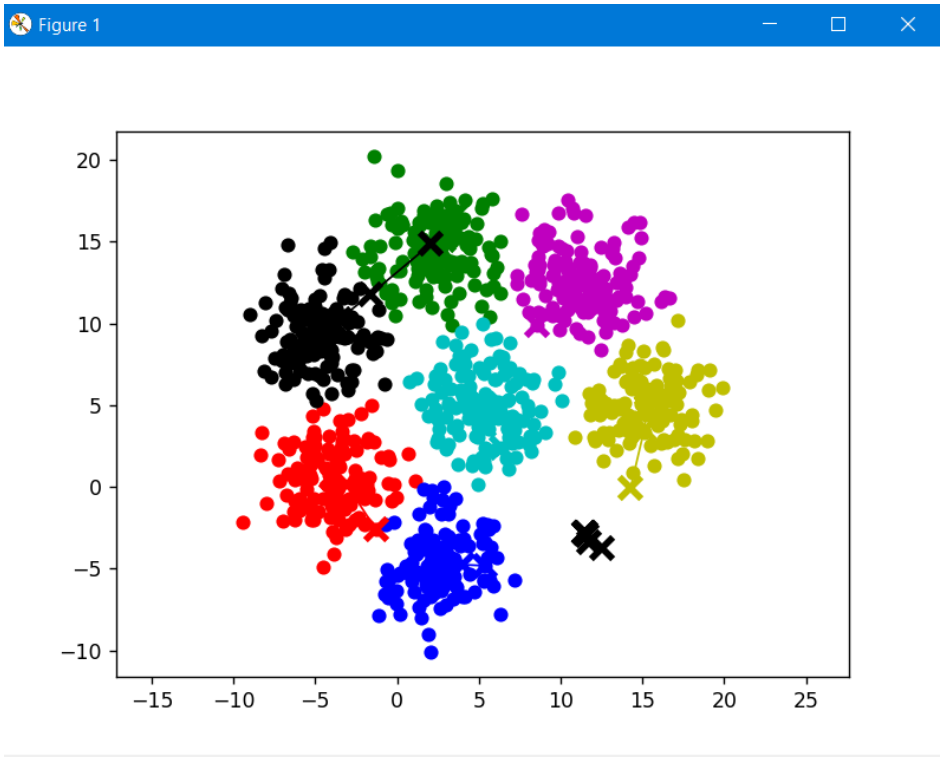
```
        'Number of iterations till termination: {}'.format(len(all_means) - 1)
        print
        'Convergence achieved: {}'.format(hasconverged(all_means[-1],
all_means[-2]))

        if verbose:
            print
            '\nFinal means:'
            print
            all_means[-1]
            print
            ''

    print
    '\n\nAverage SSE: {}'.format(float(totalSSE) / args['numexperiments'])
    print
    'Average number of iterations: {}'.format(float(totaliter) /
args['numexperiments'])

    if args['numexperiments'] == 1:
        # save the result
        writefile(args['output'], all_means[-1])

        # If the data is 2-d and small, visualize it.
        if len(data) < 5000 and len(data[0]) == 2:
            visualize_data(data, all_means, args)

        visualize_performance(data, all_means, args['dist'])
```

Task 1 :
Implement all of the following 7 functions in kmeans.py.
distance_euclidean(p1, p2)
distance_manhattan(p1, p2)
initialization_forgy(data, k)
iteration_one(data, means, distance)
hasconverged(old_means, new_means, epsilon)
iteration_many(data, means, distance, numiter, epsilon)
performance_SSE(data, means, distance)
Test your code by running this command.

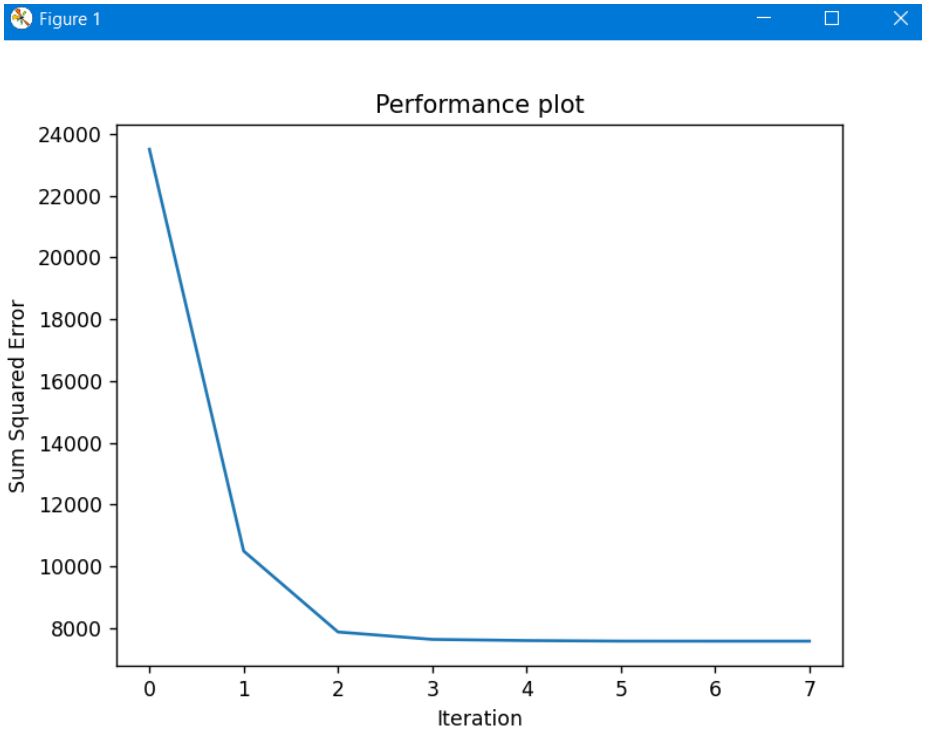
python kmeans.py --input datasets/flower.csv --iter 100 --epsilon 1e-3 --init forgy --dist euclidean --k 8 --seed $RANDOM
Try different values of k, and try both Euclidean and Manhattan distances.

All the given functions are implemented above in the code.

Clustering on flowers dataset.



We can see that total 7 clusters are formed.



The error remains fairly constant after third iteration.

Task 2: Testing and Performance (3 marks)
Test your code on the following data sets.
datasets/100.csv: Use k=2, numexperiments =100
datasets/1000.csv : Use k=5, numexperiments =25
datasets/10000.csv : Use k=20, numexperiments =10
Use epsilon=10−2 and the Euclidean distance metric for every experiment. Here is an example.

python kmeans.py --epsilon 1e-2 --init forgy --dist euclidean --input datasets/100.csv --k 2 -numexperiments 100

1. For each dataset and distance metric, report “average SSE" and “average iterations”. (1.5 marks)
Answer:

Dataset	Average SSE	Average Iterations
100.csv		
1000.csv		
10000.csv		

2. Run your code on datasets/garden.csv, with different values of k. Looking at the performance plots, does the SSE of k-means algorithm ever increase as the iterations are made? (1 mark)
Answer: No the SSE never increases in the k-means algorithm. In fact it can be proven that it will never increase. The decrease happens at two points, one is when we change the label of the points according to the centroid closest to it. This means that the closest centroid distance for every point that get relabelled has decreased, hence overall SSE has to decrease. Now the second step is that we get new means by making them the centroid of the current clusters. We know that the sum of squared distance is minimum from the mean (centroid), so again the SSE will decrease. And this process repeats. All this was also proved mathematically in class.

3. Look at the files 3lines.png and mouse.png. Manually draw cluster boundaries around the 3 clusters visible in each file (no need to submit the hand drawn clusters). Test the k-means algorithm on the datasets datasets/3lines.csv and datasets/mouse.csv. How does the algorithm’s clustering compare with the clustering you would do by hand? Why do you think this happens? (1 mark)
Answer: For 3lines dataset, I intuitively points in each line together in a cluster so as to get 3 oblong clusters. The algorithm on the other hand gave a very different answer. Ideally it should be possible to separate if the cluster centroids are in the middle of each of the three lines since then the perpendicular bisectors would mark the separating region. But this is not what we are

converging to in this case, probably because the SSE of such a situation is higher. We can also see that the centroid are close in this case, which is what we try to avoid in case of kmeans++ so this has a higher SSE. The SSE is higher because the ends of the lines are quite far from the middle and those distances add up. Since we're taking euclidean distances it is preferable that the points are in a circular space around the centroids so that most of the points in its cluster are as close to the centroid as possible.

For mouse dataset, I intuitively put the face in one cluster and each ear in one cluster. Again the algorithm doesn't match it, even though all these 3 clusters are circular. This time we see that some part of the face goes into the ear clusters. This happens because the circular region occupied by the mouse's face is large, and the ears are present close to the boundary of the face. Given the geometry of the face, the centroid for the face cluster would be somewhere close to the center of the face. Similarly for the ears. Now the points on the face near the ears are closer to the centroid of the ear than the centroid of the face due to the large radius of the face, this leads to the points getting classified with the ears.

Task 3: Implementing Random Partition and k-means++ (4 marks)
Implement all of the following functions in kmeans.py.

initialization_randompartition(data, distance, k)
initialization_kmeansplusplus(data, distance, k)
Note: You are expected to provide elaborate comments along with your code (for these 2 functions). Your marks depend on whether the TAs are able to understand your code and establish its correctness.

Test with python autograder.py 3.

Use your code by running this command.

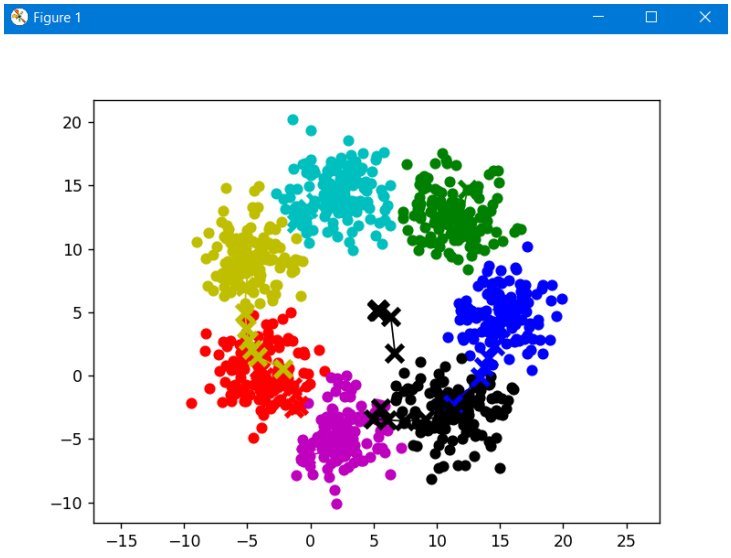
python kmeans.py --input datasets/flower.csv --epsilon 1e-2 --init kmeans++ --dist euclidean --k 8
Try both random partition and kmeans++.

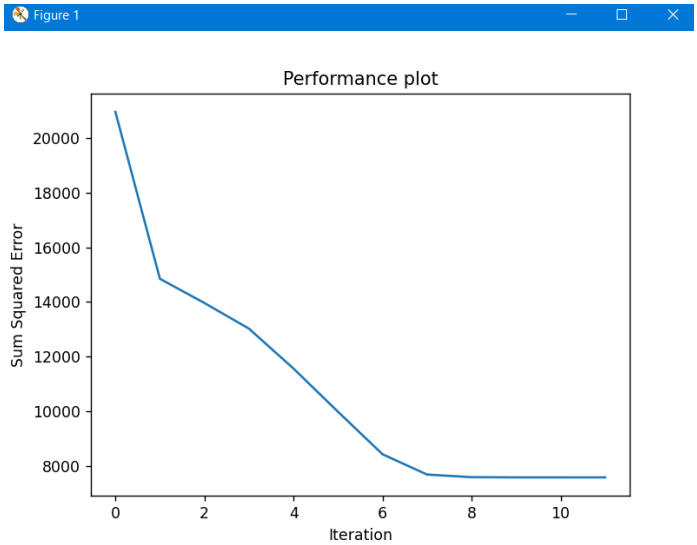
After implementing your code, test it on these data sets.

datasets/100.csv: Use k=2, numexperiments =100
datasets/1000.csv : Use k=5, numexperiments =25
datasets/10000.csv : Use k=20, numexperiments =10
Use epsilon=10^{-2} and the Euclidean distance metric for every experiment.

Run your experiments for both random partition and kmeans++ initialisation algorithms. Here is an example command.

python kmeans.py --epsilon 1e-2 --init randompartition --dist euclidean --input datasets/100.csv --k 2 -numexperiments 100





1. For each dataset, and initialization algorithm, report “average SSE” and "average iterations". (1 mark)

Answer:

Dataset	Initialization	Average SSE	Average Iterations
100.csv	RandomPartition		
100.csv	kmeans++		

1000.csv	RandomPartition	
1000.csv	kmeans++	
10000.csv	RandomPartition	
10000.csv	kmeans++	

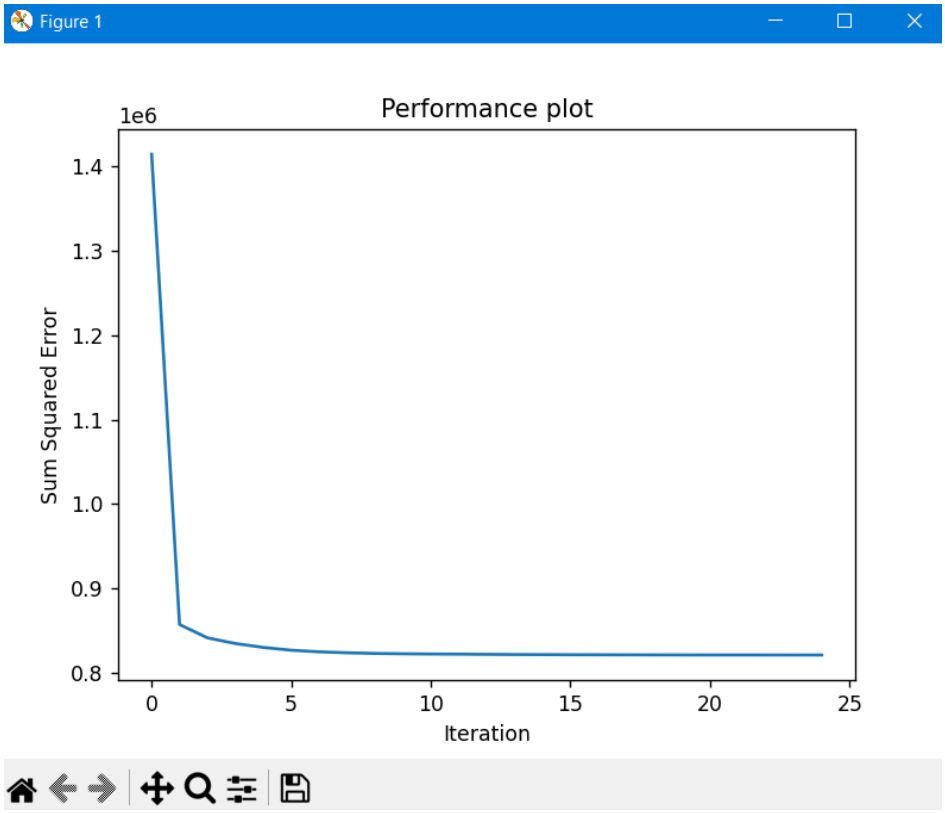
Task 4: MNIST classification (2 marks)
Template File: kmeans.py
Data set: datasets/mnist.csv

Run your algorithm on the MNIST data set as follows.

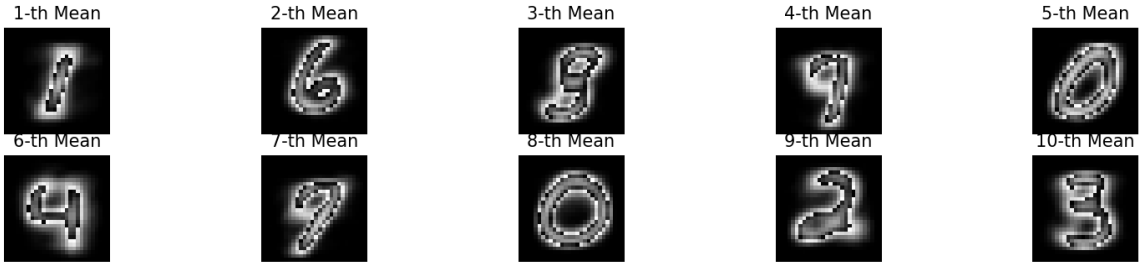
python kmeans.py --input datasets/mnist.csv --iter 100 --epsilon 1e-2 --init kmeans++ --dist euclidean --k 10 --output mnist.txt
Plot the so found cluster centres by executing this command.

python mnistplot.py mnist.txt
Look at the plots and find out a good mean for each of the 10 digits. Compile these means into the file mnistmeans.txt. You will have to run the clustering algorithm several times to get satisfactory means. Use the random seed to get different means.

File Format for mnistmeans.txt:
The i th line must contain the cluster mean for the i−1 th digit.
Each mean is represented by a comma separated list of 784 floats.



This dataset thus requires at least 5 iterations.



We can observe that k means works fairly well on MNIST dataset.

Conclusions :

- k-means has trouble clustering data where clusters are of varying sizes and density. To cluster such data, you need to generalize k-means.
- Centroids can be dragged by outliers, or outliers might get their own cluster instead of being ignored. Thus, k means is not robust to outliers.
- K means can also be implemented for classification in MNIST dataset. However, we saw that it did not offer a great accuracy and also misclassified a lot of images.
- Data Normalization is an important preprocessing step which ensures that each input parameter (pixel, in this case) has a similar data distribution before implementing k means.