Bharatiya Vidya Bhavan's

Sardar Patel Institute of Technology

Bhavan's Campus, Munshi Nagar, Andheri (West), Mumbai-400058-India

(Autonomous College Affiliated to University of Mumbai)

Academic SEM: VII **Year: 2022-23**

Experiment:K Means Clustering

Name:	Gaurav Panchal
UID:	2019120046
Class:	BE EXTC
Batch:	B
Subject:	Data Analytics Lab

Objective: The focus of this lab is k-means clustering. We will look at the vanilla algorithm, its performance, and some better variants. Finally, we will use clustering for classifying the MNIST data set.

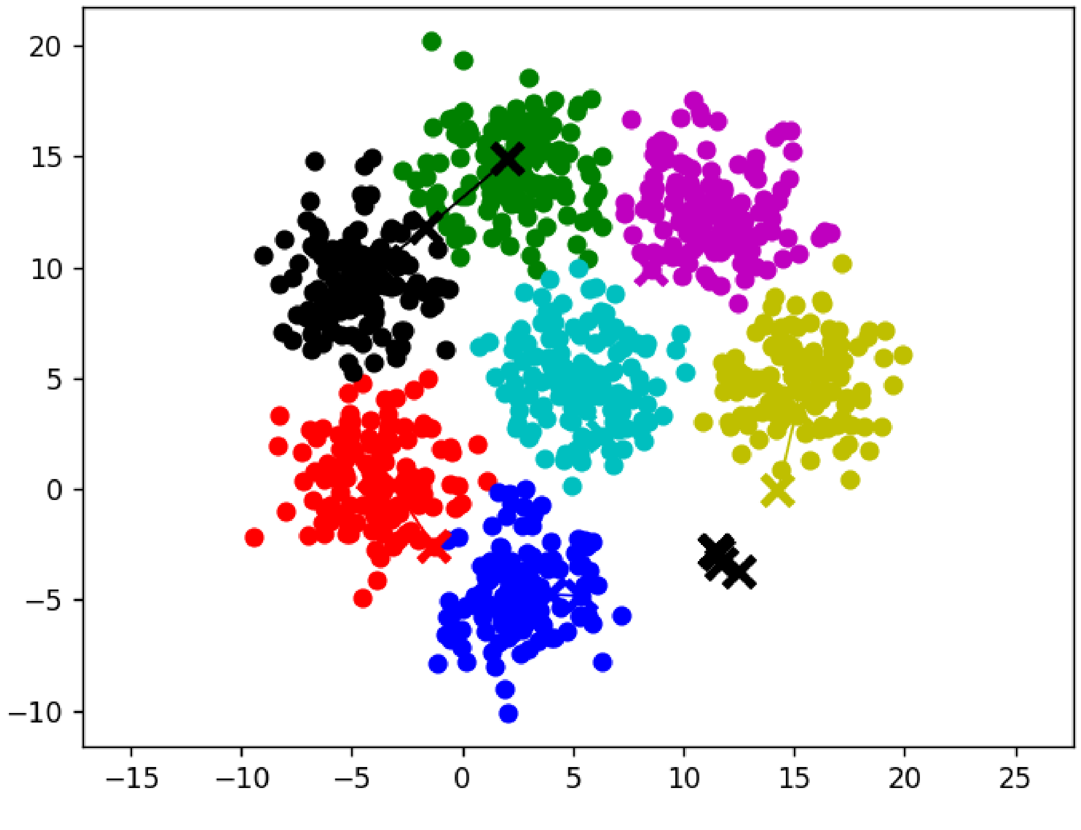
System Requirements: Python 3.9

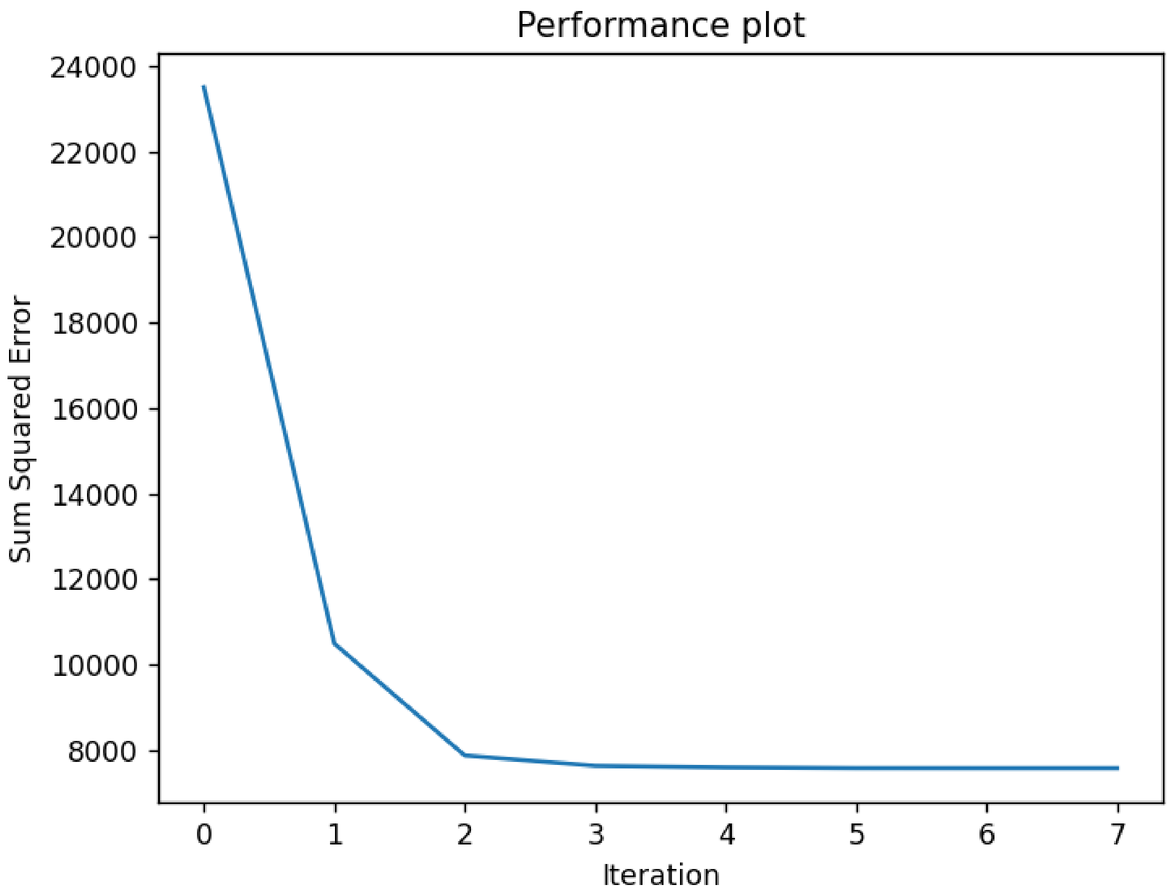
Code:
```
def distance_euclidean(p1, p2):
    '''
    p1: tuple: 1st point
    p2: tuple: 2nd point
    Returns the Euclidean distance b/w the two points.
    '''

    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 = []

    # 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.

    # The first centroid is randomly selected
    means.append(random.sample(data, 1)[0])
    # Current minimum distance squared from any centroid
    # The probability with which to select a point will be 
proportional to this
    min_dist = [float('Inf')] * len(data)

    # Going over means; adding 1 new mean in each loop
    for i in range(k-1):
        # Updating minimum for all points
        for j in range(len(data)):
            # Calculate minimum distance
            d = distance(means[i], data[j])
            # Squaring the distance since the probability will be 
proportional to this.
            d = d*d
            if d < min_dist[j]:
                min_dist[j] = d

        # Normalising min_dist array to get an array of 
probabilities
        s = sum(min_dist)
        prob = [i/s for i in min_dist]
        # Choose a random index with probabilities proportional to 
min_dist
        idx = np.random.choice(len(data), 1, p=prob)[0]
        # Append the chosen point to means
        means.append(data[idx])
        # Note that in the next iteration, the min_dist for points 
in `means` will become 0
        # So they will not be chosen again

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

def initialization_randompartition(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 = []
    means.append(random.sample(data, 1)[0])
    # 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.
    for i in range(k-1):
        # Updating minimum for all points
        # Choose a random index with probabilities proportional to
min_dist
        idx = np.random.choice(len(data), 1)[0]
        # Append the chosen point to means
        means.append(data[idx])
    # The first centroid is randomly selected
    #means.append(random.sample(data, 1)[0])
    # Current minimum distance squared from any centroid
    # The probability with which to select a point will be
proportional to this
    #min_dist = [float('Inf')] * len(data)
    #indices = np.random.choice(range(0, k), replace = True, size =
data.shape[0])
    # Going over means; adding 1 new mean in each loop
    #for i in range(k):
    #    means.append(data[indices == i].mean(axis=0))

    ########################################
    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
```

Results:
Task1:



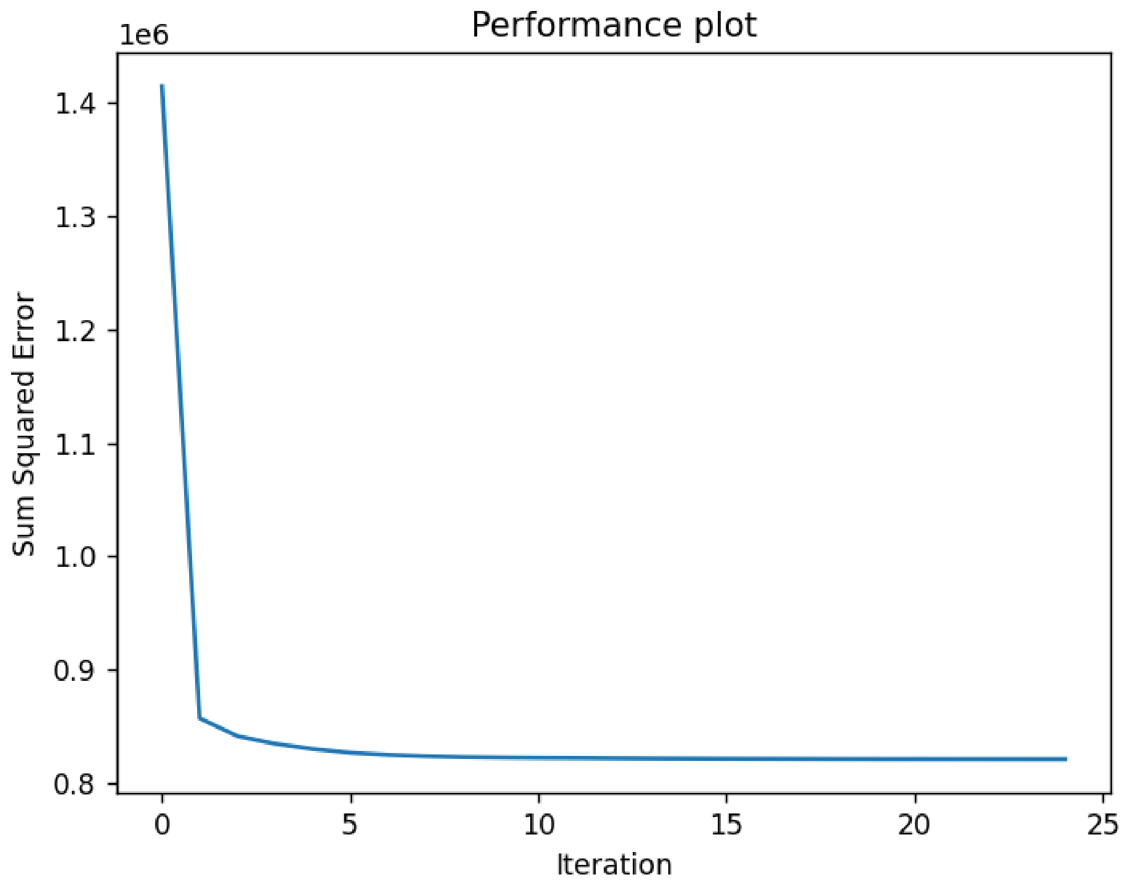


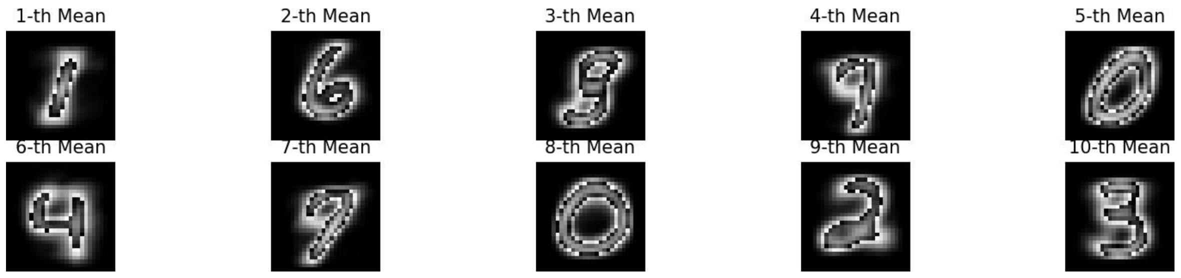
Task2:
1. Run your code on datasets/garden.csv, with different values of k. Looking at the performance plots, does the SSE of k-means algorithm ever increase as the iterations are made? (1 mark)
Answer: In the k-means method, the SSE never rises. It can even be demonstrated that it will never rise. The first time the drop occurs is when we rename the points using the centroid that is closest to them. This indicates that the distance to the centroid closest to
Every point that has been relabelled has dropped, hence the SSE must drop overall. As the centroid of the present clusters, we now move on to step two, where we obtain new means. The SSE will again drop since we know that the sum of squared distance is the least from the mean (centroid). And the cycle continues. All of this was mathematically demonstrated in class as well.

2. 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: I naturally group the three lines in the dataset together to form three oblong clusters. On the other side, the algorithm provided a very different response. If the cluster centroids are in the middle of each of the three lines, separation should be easy since the perpendicular bisectors would identify the dividing zone.

But given that the SSE of such a scenario is higher, this is not where we are convergent in this case. Additionally, as we can see, the centroid is in close proximity in this instance, which is something we strive to avoid when using kmeans++, leading to a higher SSE. The SSE is greater because the distances between the lines' endpoints and middles are greater. Since we're measuring euclidean distances, it's best if the points are arranged in a circle around the centroids, putting the majority of the points in each cluster as close as they can be to the centroid.
I instinctively grouped the face and each ear in one cluster for the mouse dataset. Even though all three of these clusters are circular, the algorithm once more cannot match it. This time, we can see that the face extends into the ear clusters in certain places. This occurs as a result of the mouse's face's enormous circumference and the presence of the ears at the face's edge. The centroid for the face cluster would be near to the face's centre given the face's geometry. The ears are the same way. Due to the enormous radius, the spots on the face near the ears are now closer to the centroid of the ear than the centroid of the face.

Task4





1. For each dataset, with kmeansplusplus initialization algorithm, report "average SSE" and "average iterations". (1 mark)
Answer:

Dataset	Initialization	Average SSE	Average Iterations
100.csv	forgy	8472	2.64
100.csv	kmeans++	8472	2.01
1000.csv	forgy	21337462	3.4
1000.csv	kmeans++	19400000	3.25
10000.csv	forgy	169946236	22.1
10000.csv	kmeans++	20783467	6.01

In every instance, the average SSE and the number of iterations are both less than forgy. This is as a result of our decision to use improved initializations for kmeans++. In kmeans, the initial centroids are chosen to be further apart from one another, making it more likely that every point will find at least one close-by centroid; in other words, for the majority of the points, the minimum distance from the initial centroids will be smaller than in the case of forgy. Therefore, the SSE is low right away. Faster convergence results from this. Additionally, better initializations increase the likelihood

Conclusion:

- K means may also be used to classify data from the MNIST dataset. However, we discovered that it was not very accurate and frequently misclassified photos.
- When there are clusters with different densities and sizes, k-means has problems clustering the data. You must generalise k-means in order to cluster such data.
- Before applying k means, data normalization is a crucial preprocessing step that makes sure
- Outliers may drag centroids or they may form their own cluster in place of being ignored. K means are therefore not resistant to outliers.