1. Class: matrix

```python

class matrix:
    def __init__(self, file_path):
        self.array_2d = None
        self.load_from_csv(file_path)

    def load_from_csv(self, file_path):   # conver to matrix in my csv file

        self.array_2d = np.loadtxt(file_path, delimiter=',')
        print("sample arrar\y_2d in file vector values:",self.array_2d[0])

    def standardise(self):   # standrdise the all data for matrix values
        mean = np.mean(self.array_2d, axis=0)
        print("mean:",mean)
        max_val = np.max(self.array_2d, axis=0)
        print("max_values:",max_val)
        min_val = np.min(self.array_2d, axis=0)
        print("min_value:",min_val)
        self.array_2d = (self.array_2d - mean) / (max_val - min_val)

    def get_distance(self, other_matrix, row_i):       # row number == row_i (row_i is input row number (vector))and matrix == other_matrix
        #Euclidean distance in specific row
        euclidean_distance = np.sqrt(np.sum((self.array_2d[row_i] - other_matrix.array_2d) ** 2, axis=1)) # axis 1 values
        return euclidean_distance

    def get_weighted_distance(self, other_matrix, weights, row_i):     # row_i is my input row
        #get Weight in Euclidean distance from specific row
        diff = self.array_2d[row_i] - other_matrix.array_2d
        weight_Euclidean_distance= np.sqrt(np.sum(weights * (diff ** 2), axis=1))
        return weight_Euclidean_distance

    def get_count_frequency(self):
        #Return the frequency of each element in the matrix
        unique, counts = np.unique(self.array_2d, return_counts=True)
        count_frequency=dict(zip(unique, counts))
        return count_frequency

```

- **__init__(self, file_path):**
    - Initializes the matrix object by loading the matrix data from the provided CSV file.
    - **Parameter:** file_path - Path to the CSV file containing matrix data.
    - Calls the load_from_csv() method.

- **load_from_csv(self, file_path)**:
    - Reads the CSV file and stores the data as a 2D NumPy array.
    - **Parameter:** file_path - Path to the CSV file.
- **standardise(self)**:
    - Standardizes the matrix values by adjusting them to have zero mean and unit variance based on the min and max values of the matrix.
    - Ensures data is normalized for clustering algorithms.
- **get_distance(self, other_matrix, row_i)**:
    - Computes the Euclidean distance between a specific row (row_i) of the current matrix and all rows of another matrix.
    - **Parameters:**
        - other_matrix - Another matrix to compare against.
        - row_i - The index of the row in the current matrix.
- **get_weighted_distance(self, other_matrix, weights, row_i)**:
    - Computes a weighted Euclidean distance between a specific row (row_i) of the current matrix and all rows of another matrix using a weights vector.
    - **Parameters:**
        - other_matrix - Another matrix to compare against.
        - weights - A vector of weights for each dimension.
        - row_i - The index of the row in the current matrix.
- **get_count_frequency(self)**:
    - Returns the frequency of each unique element in the matrix as a dictionary.
    - Useful for analyzing the distribution of elements in the matrix.

2. My Creating Functions

```
def get_initial_weights(m):
    #Generate initial random weights
    weights = np.random.rand(m)
    initial_weight=weights / np.sum(weights)
    return initial_weight
```

- **get_initial_weights(m)**:
    - Generates a random initial weights vector for m dimensions.
    - The weights sum up to 1, which is used in weighted distance calculations.

- **get_centroids(data, S, K)**:

```python
def get_centroids(data, S, K):
    #Compute centroids for clusters
    centroids = np.zeros((K, data.array_2d.shape[1]))
    for k in range(K):
        rows_in_cluster = data.array_2d[S == k]
        if len(rows_in_cluster) > 0:
            centroids[k] = np.mean(rows_in_cluster, axis=0)
    return centroids
```

  - Computes the centroids for K clusters based on the current cluster assignments S.
  - **Parameters:**
    - data - The matrix data.
    - S - A vector of cluster assignments for each row.
    - K - The number of clusters.

- **get_separation_within(data, centroids, S, K)**:

```python
def get_separation_within(data, centroids, S, K):
    #Calculate separation within clusters
    a = np.zeros(data.array_2d.shape[1])
    for j in range(data.array_2d.shape[1]):
        for k in range(K):
            rows_in_cluster = data.array_2d[S == k]
            a[j] += np.sum(np.linalg.norm(rows_in_cluster[:, j] -
centroids[k, j]))
    return a
```

  - Calculates the within-cluster separation by computing the norm between elements in a cluster and their respective centroids.

- **get_separation_between(data, centroids, S, K)**:

```python
def get_separation_between(data, centroids, S, K):
   #Calculate separation between clusters
```

```
    b = np.zeros(data.array_2d.shape[1])
    for j in range(data.array_2d.shape[1]):
        for k in range(K):
            count_k = np.sum(S == k)
            b[j] += count_k * np.linalg.norm(centroids[k, j] -
np.mean(data.array_2d[:, j]))
    return b

```

- Calculates the between-cluster separation, computing the norm between centroids and the mean of the data.

- **get_groups(data, K):**

```

def get_groups(data, K):
    # Ensure that K is not greater than the number of data points
    num_rows, num_cols = data.array_2d.shape

    if K > num_rows:
        raise ValueError(f"Number of clusters K={K} cannot be greater
than the number of data points {num_rows}.")

    # Initialize group assignments (S) and centroids
    S = np.zeros(num_rows, dtype=int)
    centroids = data.array_2d[np.random.choice(num_rows, K,
replace=False)]

    print(f"Initial centroids shape: {centroids.shape}")

    while True:
        new_S = np.zeros(num_rows, dtype=int)
        for i in range(num_rows):
            distances = np.linalg.norm(data.array_2d[i] - centroids,
axis=1)
            # print(f"Row {i} distances to centroids: {distances}")
            new_S[i] = np.argmin(distances)
            # print(f"Assigned cluster for row {i}: {new_S[i]}")

        if np.all(S == new_S):
            break

        S = new_S
        print(f"Updated cluster assignments: {S}")

        # Recompute centroids based on new group assignments
        for k in range(K):
```

```
            cluster_points = data.array_2d[S == k]
            # print(f"Points in cluster {k}: {cluster_points}")

            if len(cluster_points) > 0:  # Avoid empty clusters
                centroids[k] = np.mean(cluster_points, axis=0)
                # print(f"Updated centroid {k}: {centroids[k]}")
            else:
                # print(f"Cluster {k} is empty. Reassigning to random
point.")
                centroids[k] =
data.array_2d[np.random.choice(num_rows)]


    return S

# def get_groups(data, K):
#     #Assign groups based on the nearest centroids
#     S = np.zeros(data.array_2d.shape[0], dtype=int)
#     centroids =
data.array_2d[np.random.choice(data.array_2d.shape[0], K,
replace=False)]
#     while True:
#         new_list=[]
#         for c in centroids:
#             norm_=np.argmin(np.linalg.norm(data.array_2d - c,
axis=1))
#             new_list.append(norm_)
#         print(new_list)
#         new_S = np.array(new_list)
#         if np.all(S == new_S):
#             break
#         S = new_S
#     return S

```

- Assigns each row of the matrix to one of K clusters based on the closest centroids. Repeats the process until the assignments stabilize.

- **get_new_weights(data, centroids, weights, S, K)**:

```

def get_new_weights(data, centroids, weights, S, K):
    #Update the weights vector
    a = get_separation_within(data, centroids, S, K)
    b = get_separation_between(data, centroids, S, K)
    new_weights = 0.5 * (weights + (b / a) / np.sum(b / a))
    return new_weights

```

- Updates the weights for each dimension based on the separation within and between clusters.

- **Our documentation text function :**

```
# Test function with file path
def run_test(file_path):
    #Run tests in custom file path
    m = matrix(file_path)
    m.standardise()
    for k in range(2, 11):
        for i in range(20):
            S = get_groups(m, k)
            print(f'K={k}, Frequency: {m.get_count_frequency()}')
file_path = "E:\\anubavam\\Data (2).csv"

run_test(file_path)

```

3. Functions

- **test_standardisation(file_path)**:

```
def test_standardisation(file_path):
    m = matrix(file_path)
    m.standardise()
    mean_after_standardisation = np.mean(m.array_2d, axis=0)
    print(f"Mean after standardization (should be close to 0):
{mean_after_standardisation}")
    assert np.allclose(mean_after_standardisation,
np.zeros(m.array_2d.shape[1])), "Standardization failed"

```

- Validates the standardization process by ensuring that the mean of the standardized matrix is approximately zero.

- **test_distance(file_path):**

```
def test_distance(file_path):
    m1 = matrix(file_path)
    m2 = matrix(file_path)

    # Manually compare distances
    row_index = 0
    distance = m1.get_distance(m2, row_index)
    print(f"Euclidean distance for row {row_index}: {distance}")
    assert distance is not None, "Distance calculation should not return None."
```

- Verifies the distance calculation by comparing the Euclidean distance for a given row between two matrices.

- **test_weight_update(file_path):**

```
def test_weight_update(file_path):
    m = matrix(file_path)
    m.standardise()

    initial_weights = get_initial_weights(m.array_2d.shape[1])
    S = get_groups(m, K=2)
    centroids = get_centroids(m, S, 2)
    updated_weights = get_new_weights(m, centroids, initial_weights, S, 3)
    print(f"Initial weights: {initial_weights}")
    print(f"Updated weights: {updated_weights}")

    assert np.any(updated_weights != initial_weights), "Weights should update after recalculation."

```

- Tests whether the weights update correctly after recalculating based on the separation within and between clusters.

- **test_clustering_consistency(file_path):**

```
def test_clustering_consistency(file_path):
    m = matrix(file_path)
    m.standardise()

    S1 = get_groups(m, K=4)
    S2 = get_groups(m, K=4)

    print(f"First clustering: {S1}")
    print(f"Second clustering: {S2}")

```

```python
    assert np.array_equal(S1, S2), "Clustering should be consistent."
```

- Ensures that the clustering results are consistent when run multiple times with the same data.

- **4. Main Test Function: run_test(file_path):**

```python
def run_all_tests(file_path):
    test_standardisation(file_path)
    test_distance(file_path)
    test_weight_update(file_path)
    test_clustering_consistency(file_path)

file_path = "E:\\anubavam\\Data (2).csv"
run_all_tests(file_path)
```

- This function calls the standardization process and performs clustering for various values of K (number of clusters).
- It runs tests by printing cluster assignment frequencies for each K value from 2 to 10.