1. **Class: matrix**
2. class matrix:
3. def \_\_init\_\_(self, file\_path):
4. self.array\_2d = None
5. self.load\_from\_csv(file\_path)
6. def load\_from\_csv(self, file\_path):   # conver to matrix in my csv file
8. self.array\_2d = np.loadtxt(file\_path, delimiter=',')
9. print("sample arrar\y\_2d in file vector values:",self.array\_2d[0])
10. def standardise(self):   # standrdise the all data for matrix values
11. mean = np.mean(self.array\_2d, axis=0)
12. print("mean:",mean)
13. max\_val = np.max(self.array\_2d, axis=0)
14. print("max\_values:",max\_val)
15. min\_val = np.min(self.array\_2d, axis=0)
16. print("min\_value:",min\_val)
17. self.array\_2d = (self.array\_2d - mean) / (max\_val - min\_val)
18. def get\_distance(self, other\_matrix, row\_i):       # row number == row\_i (row\_i is input row number (vector))and matrix == other\_matrix
19. #Euclidean distance in specific row
20. euclidean\_distance = np.sqrt(np.sum((self.array\_2d[row\_i] - other\_matrix.array\_2d) \*\* 2, axis=1)) # axis 1 values
21. return euclidean\_distance
22. def get\_weighted\_distance(self, other\_matrix, weights, row\_i):     # row\_i is my input row
23. #get Weight in Euclidean distance from specific row
24. diff = self.array\_2d[row\_i] - other\_matrix.array\_2d
25. weight\_Euclidean\_distance= np.sqrt(np.sum(weights \* (diff \*\* 2), axis=1))
26. return weight\_Euclidean\_distance
27. def get\_count\_frequency(self):
28. #Return the frequency of each element in the matrix
29. unique, counts = np.unique(self.array\_2d, return\_counts=True)
30. count\_frequency=dict(zip(unique, counts))
31. 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)**:
* 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)**:
* 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)**:
* 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}")
* 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):**
* 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.