```
'''ASSIGNMENT: IDENTIFYING GROUPS OF SIMILAR WINES'''

# Importing libraries
import numpy as np
import pandas as pd


# Initializing class
class Matrix:
    # Initializing the Matrix object with a 2D array
    def __init__(self, array_2d):

        self.array_2d = np.array(array_2d)  # Convert the input data to a
numpy array
        self.rows, self.cols = self.array_2d.shape  # Get the shape of the
array

    @staticmethod
    # Loading data from a CSV file
    def load_from_csv(file_name):

        data_frame = pd.read_csv(file_name)  # Read the CSV file using
pandas
        return Matrix(data_frame.values)  # Convert it to a Matrix object


    # Standardise the matrix data using the formula:
    # D'_ij = (D_ij - mean(D_j)) / (max(D_j) - min(D_j))
    def standardise(self):

        standardised_data = np.zeros_like(self.array_2d, dtype=float)  #
Initializing an array for standardized values
        for j in range(self.cols):  # Iterating through each column
            column = self.array_2d[:, j]  # Extracting column data
            mean_col = np.mean(column)  # Calculate mean of the column
            max_col = np.max(column)  # Getting the maximum value in the
column
            min_col = np.min(column)  # Getting the minimum value in the
column
            if max_col - min_col != 0:  # Avoiding division by zero
                standardised_data[:, j] = (column - mean_col) / (max_col -
min_col)  # Standardise
        return Matrix(standardised_data)  # Returning a new Matrix with
the standardized data


    # Calculating the Euclidean distance between row i of this matrix and
each row in another matrix.
    # Returns the distances as a column vector.
    def get_distance(self, other_matrix, row_i):

        row_i_data = self.array_2d[row_i]  # Getting the row_i data from
the matrix
```

```
        distances = np.sqrt(np.sum((other_matrix.array_2d - row_i_data) **
2, axis=1))  # Calculateing Euclidean distance
        return distances.reshape(-1, 1)  # Returning the distances as a
column vector


    # Calculating the Weighted Euclidean distance between row i of this
matrix and each row in another matrix.
    # The weights array is applied to each dimension
    def get_weighted_distance(self, other_matrix, weights, row_i):

        row_i_data = self.array_2d[row_i]  # Getting the row_i data from
the matrix
        weighted_distances = np.sqrt(np.sum(weights *
(other_matrix.array_2d - row_i_data) ** 2, axis=1))  # Calculateing
Weighted distance
        return weighted_distances.reshape(-1, 1)  # Returning the
distances as a column vector


    # Counting the frequency of unique elements in the matrix and return a
dictionary
    def get_count_frequency(self):

        unique, counts = np.unique(self.array_2d, return_counts=True)  #
Getting unique values and their counts
        return dict(zip(unique, counts))  # Returning the results as a
dictionary


# Functions outside the class

# Initializing a weight vector of length m where the sum of the weights is
1
def get_initial_weights(m):

    weights = np.random.rand(m)  # Generating random weights
    return weights / np.sum(weights)  # Normalizing the weights to sum to
1


# Computing centroids for K clusters. Each centroid is the mean of all
rows assigned to a cluster.
# Returning the centroids as a Matrix object.
def get_centroids(matrix, S, K):

    centroids = np.zeros((K, matrix.cols))  # Initializing a matrix for
centroids
    for k in range(K):
        cluster_rows = matrix.array_2d[S == k]  # Getting all rows
assigned to cluster k
        if len(cluster_rows) > 0:
            centroids[k] = np.mean(cluster_rows, axis=0)  # Computing the
mean of the rows
```

```
    return Matrix(centroids)


#Calculating the within-cluster separation for each dimension.
def get_separation_within(matrix, centroids, S, K):

    m = matrix.cols  # Number of dimensions
    separation_within = np.zeros((1, m))  # Initializing the separation
matrix
    for j in range(m):
        for i in range(matrix.rows):
            u_ik = 1 if S[i] == j else 0  # Indicator for cluster
assignment
            centroid_k = centroids.array_2d[S[i], j]  # Getting the
centroid of the cluster
            separation_within[0, j] += u_ik *
np.linalg.norm(matrix.array_2d[i, j] - centroid_k) ** 2  # Calculating
within-cluster separation
    return separation_within


# Calculate the between-cluster separation for each dimension
def get_separation_between(matrix, centroids, S, K):

    m = matrix.cols  # Number of dimensions
    separation_between = np.zeros((1, m))  # Initializing the separation
matrix
    for j in range(m):
        for k in range(K):
            Nk = np.sum(S == k)  # Counting the number of rows in cluster
k
            if Nk > 0:
                separation_between[0, j] += Nk *
np.linalg.norm(centroids.array_2d[k, j] - matrix.array_2d[:, j].mean()) **
2  # Calculating between-cluster separation
    return separation_between


# Assigning each row in the matrix to a random cluster, creating the group
assignment array S.
def get_groups(matrix, K):

    S = np.random.randint(0, K, size=matrix.rows)  # Random initialization
of cluster assignments
    return S


# Updating the weights based on the separation within and between
clusters.
def get_new_weights(centroids, separation_within, separation_between,
old_weights, S, K):

    new_weights = np.zeros_like(old_weights)  # Initializing the new
weights array
```

```
    for j in range(len(old_weights)):
        sum_term = np.sum(separation_between[0, j] / separation_within[0,
j] for j in range(len(old_weights)))  # Summation term in the weight
update formula
        new_weights[j] = 0.5 * (old_weights[j] + (separation_between[0, j]
/ (separation_within[0, j] * sum_term)))  # Updating each weight
    return new_weights


# Test run function

# Testing the algorithm for printing the frequency of unique elements from
provided data.
def run_test():

    m = Matrix.load_from_csv('Data (2).csv')  # Loading the matrix from a
CSV file
    for k in range(2, 11):  # Looping over different numbers of clusters
        for i in range(20):  # Performing 20 iterations for each cluster
size
            S = get_groups(m, k)  # Randomly assigning groups
            print(f"Clusters: {k}, Frequency: {m.get_count_frequency()}")
# Printing the result


# Example usage :
file_path = r"C:\Users\rahul\Jupyter
Documents\Identifying_groups_of_similar_wines_Python_Task_Anubavam\Data
(2).csv"  # Specify the path to your CSV file
data_matrix = Matrix.load_from_csv(file_path)  # Load data into the Matrix
object

# Run the test
run_test()  # Execute the test function



# Conclusion:

# The code provides a framework for implementing and testing clustering
algorithms, with a focus on:
# 1.Standardizing data.
# 2.Calculating distances (Euclidean and weighted).
# 3.Performing clustering operations such as random group assignment,
calculating centroids, and updating weights based on separation measures.
```

```
Python 3.11.5 | packaged by Anaconda, Inc. | (main, Sep 11 2023, 13:26:23) [MSC v.1916 64
bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 8.15.0 -- An enhanced Interactive Python.

Restarting kernel...
```

```
In [1]:         'C:/Users/rahul/Jupyter Documents/
Identifying_groups_of_similar_wines_Python_Task_Anubavam/
Identifying_groups_of_similar_wines.py'      ='C:/Users/rahul/Jupyter Documents/
Identifying_groups_of_similar_wines_Python_Task_Anubavam'
Clusters: 2, Frequency: {0.13: 1, 0.14: 2, 0.17: 5, 0.19: 2, 0.2: 2, 0.21: 6, 0.22: 6,
0.24: 7, 0.25: 2, 0.26: 11, 0.27: 8, 0.28: 4, 0.29: 10, 0.3: 8, 0.31: 2, 0.32: 9, 0.33: 1,
0.34: 9, 0.35: 1, 0.37: 8, 0.39: 5, 0.4: 8, 0.41: 2, 0.42: 6, 0.43: 11, 0.44: 1, 0.45: 3,
0.47: 6, 0.48: 6, 0.49: 1, 0.5: 7, 0.51: 1, 0.52: 6, 0.53: 7, 0.54: 1, 0.55: 4, 0.56: 4,
0.57: 6, 0.58: 8, 0.59: 2, 0.6: 9, 0.61: 6, 0.62: 2, 0.63: 5, 0.64: 3, 0.65: 2, 0.66: 5,
0.67: 2, 0.68: 4, 0.69: 2, 0.7: 6, 0.72: 1, 0.73: 4, 0.74: 3, 0.75: 6, 0.76: 3, 0.77: 1,
0.78: 3, 0.79: 2, 0.8: 5, 0.81: 2, 0.82: 2, 0.83: 5, 0.84: 3, 0.85: 1, 0.86: 5, 0.87: 2,
0.88: 4, 0.89: 6, 0.9: 2, 0.906: 1, 0.91: 3, 0.92: 6, 0.93: 3, 0.94: 7, 0.95: 4, 0.96: 7,
0.97: 2, 0.98: 5, 0.99: 3, 1.0: 2, 1.01: 3, 1.02: 5, 1.03: 5, 1.04: 10, 1.05: 5, 1.06: 4,
1.07: 5, 1.08: 2, 1.09: 7, 1.1: 6, 1.11: 2, 1.12: 6, 1.13: 5, 1.14: 3, 1.15: 5, 1.16: 3,
1.17: 2, 1.18: 1, 1.19: 5, 1.2: 2, 1.21: 1, 1.22: 3, 1.23: 7, 1.24: 3, 1.25: 15, 1.26: 1,
1.27: 2, 1.28: 7, 1.29: 4, 1.3: 4, 1.31: 4, 1.32: 1, 1.33: 5, 1.34: 2, 1.35: 13, 1.36: 8,
1.37: 1, 1.38: 6, 1.39: 4, 1.4: 5, 1.41: 4, 1.42: 4, 1.43: 2, 1.44: 2, 1.45: 5, 1.46: 8,
1.47: 2, 1.48: 7, 1.5: 4, 1.51: 6, 1.52: 1, 1.53: 4, 1.54: 3, 1.55: 4, 1.56: 7, 1.57: 4,
1.58: 3, 1.59: 5, 1.6: 5, 1.61: 6, 1.62: 7, 1.63: 7, 1.64: 6, 1.65: 7, 1.66: 6, 1.67: 5,
1.68: 9, 1.69: 4, 1.7: 7, 1.71: 5, 1.72: 4, 1.73: 8, 1.74: 3, 1.75: 9, 1.76: 4, 1.77: 5,
1.78: 3, 1.79: 2, 1.8: 5, 1.81: 5, 1.82: 6, 1.83: 5, 1.84: 2, 1.85: 3, 1.86: 4, 1.87: 8,
1.88: 3, 1.89: 2, 1.9: 8, 1.92: 8, 1.93: 2, 1.94: 2, 1.95: 9, 1.96: 2, 1.97: 4, 1.98: 10,
1.99: 3, 2.0: 10, 2.01: 4, 2.02: 3, 2.03: 5, 2.04: 3, 2.05: 5, 2.06: 5, 2.08: 6, 2.09: 1,
2.1: 7, 2.11: 3, 2.12: 6, 2.13: 4, 2.14: 5, 2.15: 4, 2.16: 3, 2.17: 5, 2.18: 1, 2.19: 4,
2.2: 14, 2.21: 4, 2.22: 2, 2.23: 4, 2.24: 3, 2.25: 3, 2.26: 7, 2.27: 6, 2.28: 8, 2.29: 5,
2.3: 10, 2.31: 6, 2.32: 7, 2.33: 1, 2.34: 1, 2.35: 5, 2.36: 8, 2.37: 2, 2.38: 8, 2.39: 3,
2.4: 8, 2.41: 4, 2.42: 7, 2.43: 3, 2.44: 3, 2.45: 12, 2.46: 6, 2.47: 1, 2.48: 8, 2.49: 1,
2.5: 12, 2.51: 5, 2.52: 3, 2.53: 5, 2.54: 2, 2.55: 5, 2.56: 5, 2.57: 3, 2.58: 4, 2.59: 3,
2.6: 13, 2.61: 5, 2.62: 6, 2.63: 3, 2.64: 5, 2.65: 12, 2.67: 3, 2.68: 6, 2.69: 4, 2.7: 11,
2.71: 1, 2.72: 4, 2.73: 2, 2.74: 6, 2.75: 4, 2.76: 5, 2.77: 3, 2.78: 6, 2.79: 2, 2.8: 9,
2.81: 5, 2.82: 2, 2.83: 3, 2.84: 3, 2.85: 7, 2.86: 4, 2.87: 6, 2.88: 4, 2.89: 2, 2.9: 6,
2.91: 4, 2.92: 3, 2.93: 2, 2.94: 2, 2.95: 6, 2.96: 6, 2.97: 1, 2.98: 4, 2.99: 3, 3.0: 13,
3.02: 2, 3.03: 5, 3.04: 1, 3.05: 4, 3.07: 1, 3.08: 2, 3.1: 6, 3.12: 3, 3.13: 2, 3.14: 2,
3.15: 3, 3.16: 2, 3.17: 8, 3.18: 4, 3.19: 2, 3.2: 3, 3.21: 3, 3.22: 2, 3.23: 2, 3.24: 2,
3.25: 5, 3.26: 3, 3.27: 4, 3.28: 2, 3.29: 1, 3.3: 8, 3.31: 1, 3.32: 1, 3.33: 3, 3.35: 2,
3.36: 1, 3.37: 2, 3.38: 4, 3.39: 3, 3.4: 7, 3.43: 2, 3.44: 1, 3.45: 2, 3.47: 1, 3.48: 1,
3.49: 1, 3.5: 2, 3.52: 3, 3.53: 1, 3.54: 1, 3.55: 2, 3.56: 2, 3.57: 2, 3.58: 4, 3.59: 3,
3.6: 1, 3.63: 1, 3.64: 2, 3.67: 1, 3.69: 2, 3.7: 2, 3.71: 1, 3.74: 3, 3.75: 1, 3.8: 5,
3.82: 1, 3.83: 1, 3.84: 2, 3.85: 2, 3.86: 1, 3.87: 1, 3.88: 2, 3.9: 2, 3.91: 1, 3.93: 2,
3.94: 1, 3.95: 1, 3.98: 1, 3.99: 1, 4.0: 2, 4.04: 1, 4.1: 2, 4.12: 1, 4.2: 1, 4.25: 1,
4.28: 2, 4.3: 1, 4.31: 1, 4.32: 1, 4.35: 1, 4.36: 2, 4.38: 1, 4.4: 1, 4.43: 1, 4.45: 1,
4.5: 3, 4.6: 5, 4.61: 1, 4.68: 1, 4.7: 1, 4.72: 1, 4.8: 2, 4.9: 2, 4.92: 1, 4.95: 1, 5.0:
3, 5.04: 2, 5.05: 1, 5.08: 1, 5.1: 3, 5.19: 1, 5.2: 1, 5.24: 1, 5.25: 1, 5.28: 1, 5.3: 1,
5.4: 3, 5.43: 1, 5.45: 1, 5.5: 1, 5.51: 1, 5.58: 1, 5.6: 3, 5.65: 2, 5.68: 1, 5.7: 3,
5.75: 2, 5.8: 1, 5.85: 1, 5.88: 1, 6.0: 2, 6.1: 1, 6.13: 1, 6.2: 2, 6.25: 1, 6.3: 1, 6.38:
```