IDENTIFYING GROUPS OF SIMILAR WINES

1) Class matrix:

```
class matrix:

    def __init__(self, filename=None):

        # Initialize the array_2d attributes as an empty NumPy array
        self.array_2d = np.array([])

        if filename is not None:

            # Load the data from the CSV file
            self.load_from_csv(filename)

            # Standardise the data
            self.standardise()
```

This code defines a matrix class, which initializes a 2D NumPy array (array_2d). The constructor __init__() accepts an optional filename argument. If a filename is provided, it triggers the load_from_csv() function to load data from a CSV file and stores it in array_2d. Additionally, the class applies a standardise() method to standardize the loaded data, ensuring the matrix is prepared for further operations. Without a filename, the matrix remains an empty array.

(i) load_from_csv(self, filename):

```
def load_from_csv(self, filename):

    # Read CSV file using Pandas library
    df = pd.read_csv(filename, header=None)

    # Convert Pandas DataFrame to NumPy array
    self.array_2d = df.to_numpy()
```

The load_from_csv method reads data from a CSV file using the Pandas library. It takes a filename as input, reads the CSV file into a Pandas DataFrame (df) without headers, and then converts the DataFrame into a NumPy array using the to_numpy() function. This NumPy array is stored in the array_2d attribute of the matrix class for further use.

(ii) def standardise(self):

The standardise method normalizes the data in the array_2d attribute. First, it checks if array_2d is empty, displaying an error message if true. Otherwise, it iterates over each column in the matrix, calculating the mean, maximum, and minimum of each column. Using these values, the function standardizes the column by applying the formula: (value - mean) / (max - min), transforming the data to a common scale across columns.

$$D'_{ij} = \frac{D_{ij} - \overline{D_j}}{max(D_j) - min(D_j)}$$

Where:
$\overline{D_j}$ is the average of column j.
$max(D_j)$ is the highest value in column j.
$min(D_j)$ is the lowest value in column j.

```
def standardise(self):

    # Check self.array_2d is not empty
    if self.array_2d.size == 0:
        print("Error: self.array_2d is empty")
        return

    # Loop through each column of self.array_2d
    columns = self.array_2d.shape[1]

    # Loop through calculate the fomula values
    for col in range(0, columns):
        column = self.array_2d[:, col]
        mean = np.mean(column)
        max = np.max(column)
        min = np.min(column)

        # Apply Standardization formula
        self.array_2d[:, col] = (column - mean) / (max - min)
```

(iii) def get_distance(self, other_matrix, row_i):

```
def get_distance(self, other_matrix, row_i):

    # Get Specific_row from Matrix
    row = self.array_2d[row_i]

    # Initialize the List to Store the Distance
    distances = []

    # Loop through Calculate Euclidean Distance between Specific_row to All Other Rows
    for other_row in other_matrix:
        distance = ((row - other_row)**2)
        distances.append([distance])

    # Convert the list of distances to NumPy array (as a matrix with n rows and 1 column)
    return np.array(distances)
```

The get_distance method calculates the Euclidean distance between a specific row in array_2d and all rows in another matrix (other_matrix). It first retrieves the row at index row_i and initializes an empty list to store distances. For each row in other_matrix, it computes the squared Euclidean distance by subtracting the row values, squaring the result, and storing it in the distances list. Finally, the list is converted into a NumPy array and returned as a matrix with one column per distance.

Euclidean distance

The Euclidean distance between a vector **x** and a vector **y** (all vectors with size m), is given by:

$$d = \sum_{j=1}^{m} (x_j - y_j)^2$$

(iv) def get_weighted_distance(self, other_matrix, weights, row_i):

The get_weighted_distance method computes the weighted Euclidean distance between a specific row in array_2d and all rows in another matrix (other_matrix). It first retrieves the row at index row_i and initializes an empty list to store the distances. For each row in other_matrix, it calculates the weighted distance by

applying the provided weights to the squared differences between the rows and summing the result. These weighted distances are stored in the weighted_distances list, which is then converted into a NumPy array with one column and returned.

```
def get_weighted_distance(self, other_matrix, weights, row_i):

    # Get Specific_row from Matrix
    row = self.array_2d[row_i]

    # Initialize the List to Store the Distance
    weighted_distances = []

    # Loop through Calculate Euclidean Distance between Specific_row to All Other Rows
    for other_row in other_matrix:
        distance = np.sum(weights * ((row - other_row)**2))
        weighted_distances.append([distance])

    # Convert the list of weighted distance to NumPy array (as a matrix with n rows and 1 column)
    return np.array(weighted_distances)
```

Weighted Euclidean distance

There are different weighted distances, in this assignment you must follow the below. The distance between a vector **x** and a vector **y**, using the weights in a vector **w** (all three vectors with size m), given by:

$$d = \sum_{j=1}^{m} w_j (x_j - y_j)^2$$

(v) def get_count_frequency(self, S):

```
def get_count_frequency(self, S):

    # Check Cluster Matrix Output column is 1 or not
    if S.shape[1] != 1:
        return 0

    # Flatten S to make it a 1D array for easier processing
    flattened_S = S.flatten()

    # Get the Unique Values and Counts
    unique, counts = np.unique(flattened_S, return_index=True)

    # Create Dictionary Mapping Each Element with its Count
    frequency_dict = dict(zip(unique, counts))

    return frequency_dict
```

The get_count_frequency method counts the frequency of unique values in a cluster matrix S. It first checks if S has exactly one column, returning 0 if not. Then, it flattens S into a one-dimensional array for easier processing. The method uses NumPy's np.unique function to retrieve unique values and their corresponding counts. Finally, it creates a dictionary that maps each unique element to its count and returns this frequency dictionary for further analysis.

2. Functions

The functions listed below are not part of the Matrix class since they are not defined as class methods. They operate independently and are not bound to any specific instance of the class.

(i) def get_initial_weights(c):

```
def get_initial_weights(c):

    # Generate c Random Values between 0 and 1
    random_values = np.random.rand(c)

    # Normalize the Random values to make their Sum equal to 1
    normalized_weights = random_values / np.sum(random_values)

    # Reshape Matrix with 1 row and c columns
    return normalized_weights.reshape(1, c)
```

The get_initial_weights function generates initial weights for c clusters by creating c random values uniformly distributed between 0 and 1. These random values are then normalized so that their sum equals 1, ensuring they can be used as valid weights. Finally, the normalized weights are reshaped into a matrix format with one row and c columns, ready for use in clustering algorithms or other applications requiring weighted inputs.

(ii) def get_centroids(data, S, K):

```
def get_centroids(data, S, K):

    # Get the number of rows (r) and columns (c) in data
    r, c = data.shape   # (178, 13)

    # Create an Empty centroid Matrix with K rows and c columns
    centroids = np.zeros((K, c))    # Ex: K=4 means np.zeros((4,13))

    # Randomly select the K different rows from data (178 rows)
    centroids_index = np.random.choice(r, K, replace=False)   # [3,50,133,178]

    # Empty centroid Matrix updated with Random K rows matrix
    centroids = data[centroids_index]

    # Initialize Normalized Random Weights with 1 row and 13 columns
    weights = get_initial_weights(c)

    while True:

        # Store old S value before updation with new value in Centroid
        S_old = S.copy()

        # Calculate Weighted Euclidean Distance between row_i and all centroids
        for i in range(0, r):
            distances_to_centroids = m.get_weighted_distance(centroids, weights, i)
```

```
        # Find the index of the centroid with the minimum distance
        closest_centroid_index = np.argmin(distances_to_centroids)

        # Update the S matrix with the index of the closest centroid
        S[i,0] = closest_centroid_index
```

```
    # Check if S not updated means Clustering perfectly and Break the Loop
    if np.array_equal(S, S_old):
        break

    # Updating the Centroid Position based Recalculation
    for k in range(0, K):

        # Selecting the Assigned Rows for Cluster k ---> Ex k=1 means select all rows of cluster '1' like 60 rows
        assigned_rows = data[S.flatten() == k]

        # If Data Points connected to Centroid means calculate mean to move Centroid position
        if len(assigned_rows) > 0:
            centroids[k] = np.mean(assigned_rows, axis=0)
```

```
    # Updating the Weights based Recalculation
    weights = get_new_weights(data, centroids, weights, S, K)

return S
```

The get_centroids function calculates centroids for clustering data. It first retrieves the dimensions of the input data (data) and initializes an empty centroid matrix with K rows and the same number of columns as data. Random rows from data are selected as initial centroids, and normalized random weights are generated for distance calculations.

The function enters a loop where it continuously updates the cluster assignments (S). For each data point, it computes the weighted Euclidean distance to all centroids and assigns each point to the closest centroid. If no changes occur in S, indicating stable clustering, the loop breaks. Otherwise, the centroids are updated by calculating the mean of the data points assigned to each centroid, and weights are recalculated accordingly. Finally, the updated cluster assignments are returned.

(iii) def get_separation_within(data, centroids, S, K):

```
def get_separation_within(data, centroids, S, K):

    # Get the number of rows (r) and columns (c) in data
    r,c = data.shape

    # Initialize the separation within clusters matrix with 1 row and c columns
    a = np.zeros((1, c))

    # Loop through Cluster
    for k in range(0,K):
        # Loop through Each Row in Data
        for i in range(0,r):

            # Check if the current row i is assigned to cluster k (Uik = 1)
            if S[i,0] == k:

                # Calculate the Euclidean Distance from Row_i to k-th Centroid
                distance = m.get_distance(centroids, i)

                # Accumulate the squared distance
                a += distance[k,0]
    return a
```

The get_separation_within function computes the within-cluster separation for a dataset by accumulating the squared Euclidean distances between data points and their corresponding centroids. It iterates through each cluster and each assigned data point, calculating the distance from each point to its centroid. These distances are summed up in a matrix a, which represents how tightly grouped the points within each cluster are around their centroid. The function ultimately returns this matrix, providing a measure of cluster cohesion.

Separation within clusters
Let us call the vector **a** the separation within clusters. You should implement this vector as a matrix containing 1 row and m columns. For each j=1, 2, ..., m:

$$a_j = \sum_{k=1}^{K} \sum_{i=1}^{n} u_{ik} d(D'_{ij}, c_{kj}),$$

Where:
u_{ik} is equal to one if the row i in **S** is equal to k, and zero otherwise.
c_{kj} is the value at row k and column j of the matrix containing the centroids.
$d(D'_{ij}, c_{kj})$ is the Euclidean distance between D'_{ij} and c_{kj}.

(iv) def get_separation_between(data, centroids, S, K):

```
def get_separation_between(data, centroids, S, K):

    # Get the number of rows (r) and columns (c) in data
    r,c = data.shape

    # Initialize the separation between clusters matrix with 1 row and 1 column
    b = np.zeros((1, c))

    # Calculate the overall mean of the dataset for each feature (1 row, c columns)
    overall_mean = np.mean(data, axis=0)

    # Loop through each cluster (k = 0 to K-1)
    for k in range(0,K):

        # Find count of rows are assigned to cluster k (Nk)
        N_k = np.sum(S == k)

        # Loop through each Feature
        for j in range(0,c):

            # Calculate the Euclidean distance
            Distance = (centroids[k,j] - overall_mean[j])**2

            # Accumulate the Separation Value
            b[0,j] += N_k * Distance

    return b
```

The get_separation_between function calculates the separation between clusters by assessing how far each centroid is from the overall mean of the dataset. It starts by determining the number of data points assigned to each cluster and then iterates through each cluster and its features. For each feature, the squared distance between the cluster centroid and the overall mean is computed, multiplied by the number of points in that cluster, and accumulated into a matrix b. The function returns this matrix, which quantifies the separation between the clusters in relation to the overall data distribution.

Separation between clusters
Let us call the vector **b** the separation between clusters. You should implement this vector as a matrix containing 1 row and m columns. For each j=1, 2, ..., m:

$$b_j = \sum_{k=1}^{K} N_k d\left(c_{kj}, \overline{D'_j}\right),$$

Where:
N_k is the number of times the value k appears in **S**.

(v) def get_new_weights(data, centroids, weights, S, K):

```python
def get_new_weights(data, centroids, weights, S, K):

    # Get the number of rows (r) and columns (c) in data
    r, c = data.shape

    # Calculate the separation within clusters
    a = get_separation_within(data, centroids, S, K)

    # Calculate the separation between clusters
    b = get_separation_between(data, centroids, S, K)

    # Initialize New Weights Matrix with 1 row and c columns
    new_weights = np.zeros((1, c))

    # Calculate the sum of (bv / av) ---> summation(v=0 to c)
    summation_b_divide_a = np.sum(b/a)

    # Loop through Update the each weight
    for j in range(0, c):

        b_divide_a = b[0,j] / a[0,j]

        new_weights[0,j] += 0.5 * (weights[0,j] + (b_divide_a / summation_b_divide_a))

    return new_weights
```

The get_new_weights function updates the weights used for calculating distances in a clustering algorithm by considering the separation within and between clusters. It first computes the within-cluster separation (a) and the between-cluster separation (b). Then, it initializes a new weights matrix and calculates the summation of the ratios of b to a. For each feature, the function updates the weights by averaging the current weight with a ratio of the between-cluster to within-cluster separation, normalized by the total separation. The updated weights, which enhance the clustering process, are then returned.

Calculating weights

Let the vectors **a** and **b** be calculated as per above. Let **w** be the old weight vector, the values of the new weight vector **w'** for j=1, 2, …, m is:

$$w'_j = \frac{1}{2}\left(w_j + \frac{b_j/a_j}{\sum_{v=1}^{m}(b_v/a_v)}\right)$$

(vi) def get_groups(data, K):

```python
def get_groups(data, K):

    # Number of Rows from data
    r = data.shape[0]    # 178

    # Initialize Matrix S with r rows and 1 column
    S = np.zeros((r,1))

    # Get Cluster Matrix S and Centroids
    S = get_centroids(data, S, K)

    return S
```

The get_groups function identifies the cluster assignments for a given dataset based on the specified number of clusters, K. It first determines the number of rows in the dataset and initializes a cluster assignment matrix, S, with one column. The function then calls get_centroids to compute the centroids and update the cluster assignments in S. Finally, it returns updated cluster matrix, indicating which cluster each data point belongs to.

def run_test():

```
def run_test():

    global m

    # Initialize object to load CSV input file
    m = matrix('/content/Data (2).csv')

    # Loop through different Cluster values
    for k in range(2, 11):

        # Iterate multiple times to get Potential outputs
        for i in range(0, 20):

            # Find the Cluster matrix of each row
            S = get_groups(m.array_2d, k)

            # Print the Count Frequency Dict of each Cluster
            print(f'{k}={m.get_count_frequency(S)}')
```

The run_test function executes a clustering test by initializing a matrix object with data from a specified CSV file. It iterates through a range of cluster values, from 2 to 10, and for each cluster value, it performs 20 iterations to capture potential clustering outputs. Within each iteration, the function calls get_groups to obtain the cluster assignments and then prints the frequency of data points in each cluster using the get_count_frequency method. This allows for an evaluation of how the data is distributed across different clusters for varying K values.

Sample Output:

```
if __name__ == '__main__':
    run_test()
```

```
2={0.0: 59, 1.0: 0}
2={0.0: 0, 1.0: 59}
2={0.0: 0, 1.0: 4}
2={0.0: 21, 1.0: 0}
2={0.0: 59, 1.0: 0}
3={0.0: 59, 1.0: 27, 2.0: 0}
3={0.0: 62, 1.0: 0, 2.0: 59}
3={0.0: 0, 1.0: 59, 2.0: 27}
3={0.0: 59, 1.0: 21, 2.0: 0}
3={0.0: 0, 1.0: 59, 2.0: 62}
```

```
9={0.0: 67, 1.0: 147, 2.0: 136, 3.0: 59, 4.0: 0, 5.0: 70, 6.0: 149, 7.0: 4, 8.0: 62}
9={0.0: 4, 1.0: 64, 2.0: 59, 3.0: 2, 4.0: 0, 5.0: 63, 6.0: 131, 7.0: 3, 8.0: 65}
9={0.0: 144, 1.0: 3, 2.0: 62, 3.0: 59, 4.0: 4, 5.0: 0, 6.0: 11, 7.0: 63, 8.0: 2}
9={0.0: 1, 1.0: 59, 2.0: 67, 3.0: 83, 4.0: 144, 5.0: 0, 6.0: 63, 7.0: 62, 8.0: 158}
10={0.0: 0, 1.0: 147, 2.0: 144, 3.0: 60, 4.0: 43, 5.0: 59, 6.0: 134, 7.0: 62, 8.0: 83, 9.0: 2}
10={0.0: 21, 1.0: 63, 2.0: 134, 3.0: 0, 4.0: 4, 5.0: 67, 6.0: 77, 7.0: 83, 8.0: 70, 9.0: 59}
10={0.0: 149, 1.0: 0, 2.0: 63, 3.0: 135, 4.0: 67, 5.0: 59, 6.0: 62, 7.0: 4, 8.0: 2, 9.0: 121}
10={0.0: 25, 1.0: 62, 2.0: 84, 3.0: 59, 4.0: 0, 5.0: 60, 6.0: 130, 7.0: 2, 8.0: 63, 9.0: 147}
10={0.0: 62, 1.0: 43, 2.0: 64, 3.0: 59, 4.0: 0, 5.0: 4, 6.0: 63, 7.0: 3, 8.0: 60, 9.0: 1}
10={0.0: 67, 1.0: 0, 2.0: 3, 3.0: 4, 4.0: 131, 5.0: 83, 6.0: 62, 7.0: 59, 8.0: 43, 9.0: 2}
10={0.0: 70, 1.0: 109, 2.0: 69, 3.0: 60, 4.0: 59, 5.0: 63, 6.0: 1, 7.0: 0, 8.0: 2, 9.0: 65}
10={0.0: 2, 1.0: 130, 2.0: 0, 3.0: 110, 4.0: 135, 5.0: 67, 6.0: 62, 7.0: 4, 8.0: 63, 9.0: 59}
10={0.0: 0, 1.0: 65, 2.0: 62, 3.0: 2, 4.0: 21, 5.0: 11, 6.0: 73, 7.0: 59, 8.0: 67, 9.0: 4}
```