Assignment: identifying groups of similar wines

A sommelier is a trained professional who spends his or her day tasting different wines, and identifying similarities (or sometimes dissimilarities) between these. Given this is clearly an *exhausting* task, you have been hired to develop a software capable of grouping similar wines together. Your software will load a data set containing information about each wine (Alcohol content, alkalinity of ash, Proanthocyanins, colour intensity, etc) and identify which wines are similar.

Luckily, your employer has already identified a suitable algorithm and designed the software for you. All you are required to do is to write the actual source code (with comments).

Technical details:
You'll be using different data structures to accomplish the below. Your assignment must contain the code for the functions and methods below. If you wish you can write more functions and methods, but those described below must be present.
ATTENTION: in the text below any mention of matrix should be read as relating to the Class matrix you have to write (see below), and not some other matrix that may exist in some python module. Any parameters of methods or functions must be in the order presented below.

1) Class: matrix
You will code a class called matrix, which will have an attribute that must be called **array_2d**. This attribute is supposed to be a NumPy array containing numbers in two dimensions. The class matrix must have the following methods:
(in these, the parameters are in addition to `self` and must be used in the order presented below)

load_from_csv
This method should have one parameter, a file name (including, if necessary, its path and extension). This method should read this CSV file and load its data to the array_2d of matrix. Each row in this file should be a row in array_2d. Notice that in CSV files a comma separates columns (CSV = comma separated values).
You should also write code so that
```
m = matrix('validfilename.csv')
```
Creates a matrix m with the data in the file above in array_2d.

standardise
This method should have no parameters. It should standardise the array_2d in the matrix calling this method. For details on how to standardise a matrix, read the appendix.

get_distance
This method should have two parameters: a matrix (let us call it other_matrix), and a row number (let us call it row_i). This method should return a matrix containing the Euclidean distance between the row row_i of the matrix calling this method and each of the rows in other_matrix. For details about how to calculate this distance, read the appendix.
To be clear: if other_matrix has n rows, the matrix returned in this method will have n rows and 1 column.

get_weighted_distance
This method should have three parameters: two objects of matrix (let us call them other_matrix, and weights), and a row number (let us call it row_i). This method should return a matrix containing the Weighted Euclidean distance between the row row_i of the matrix calling this method and each of the rows in other_matrix, using the weights in the matrix weights. For details about how to calculate this distance, read the appendix.
To be clear: if other_matrix has n rows, the matrix returned in this method will have n rows and 1 column.

get_count_frequency
This method should have no parametes, and it should work if the array_2d of the matrix calling this method has only one column. This method should return a dictionary mapping each element of the

array_2d to the number of times this element appears in array_2d. If the number of columns in array_2d is not one, then this method should return the integer 0.

2) Functions

The code should also have the functions (i.e. not methods, so not part of the class matrix) below. No code should be outside any function or method in this assignment (the only exception are imports).

get_initial_weights

This function should have one parameter, an integer m. This function should return a matrix with 1 row and m columns containing random values, each between zero and one. The sum of these m values must be equal to one.

get_centroids

This function should have three parameters: (i) a matrix containing the data, (ii) the matrix **S**, (iii) the value of K. This function should implement the Step 9 of the algorithm described in the appendix. It should return a matrix containing K rows and the same number of columns as the matrix containing the data.

get_separation_within

This function should have four parameters: a matrix containing the data, a matrix containing the centroids, the matrix **S**, and number of groups to be created (K). This function should return a matrix with 1 row and m columns (m is the number of columns in the matrix containing the data), containing the separation within clusters (see the Appendix for details).

get_separation_between

This function should have four parameters: a matrix containing the data, a matrix containing the centroids, the matrix **S**, and number of groups to be created (K). This function should return a matrix with 1 row and m columns (m is the number of columns in the matrix containing the data), containing the separation between clusters (see the Appendix for details).

get_groups

This function should have two parameters: a matrix containing the data, and the number of groups to be created (K). This function follows the algorithm described in the appendix. It should return a matrix **S** (defined in the appendix). This function should use the other functions you wrote as much as possible. Do not keep repeating code you already wrote.

get_new_weights

This function takes five parameters: (i) a matrix containing the data, (ii) a matrix containing the centroids, (iii) a matrix containing the weights to be updated (this is referred as old weight vector in the Appendix), (iv) the matrix **S**, (v) the number of groups to be created (K). This function should return a new matrix weights with 1 row and as many columns as the matrix containing the data (and the matrix containing the centroids). This function should use the other functions you wrote as much as possible. Follow Step 10 of the algorithm in the Appendix.

run_test

Your code must contain the function below. Do not change anything, I'll have a file called Data.csv in the same folder as your code.

```
def run_test():
   m = matrix('Data.csv')
   for k in range(2,11):
        for i in range(20):
             S = get_groups(m, k)
             print(str(k)+'='+str(S.get_count_frequency()))
```

The aim of this function is just to run a series of tests. By consequence, here (and only here) you can use the hard-coded values above for the strings containing the filenames of data and values for K.

More details

You will implement a data-driven algorithm that creates groups of entities (here, an entity is a wine, described as a row in our data matrix) that are similar. If two entities are assigned to the same group by the algorithm (i.e. they have the same value in the matrix **S**), it means they are similar. This will create groups of similar wines. Your software just needs the number of groups the user wants to partition the data into, and the data itself.

The number of partitions (K) is clearly a positive integer. Your software should only allow values in the interval [2, n-1], where n is the number of rows in the data. This way you'll avoid trivial partitions.

Your software should follow the algorithm described in the appendix and generate a matrix **S** indicating to which group (1, 2, …, K) each entity (wine, a row in the data matrix) has been assigned to. Clearly **S** will have n (row) elements.

You can find more information online if you search for clustering, or k-means (this is not the algorithm we are implementing, but it is somewhat similar).

Appendix

Data standardisation

Let D be a data matrix, so that D_{ij} is the value of D at row i and column j. You can standardise D by following the equation below.

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

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.

D'_{ij} is the standardised version of D_{ij}. The algorithm below should **only** be applied to D'_{ij} (i.e. do not apply the algorithm below to D_{ij}).

Basic notation for the algorithm

n = number of rows of in the data matrix
m = number of columns in the data matrix
K = number of clusters (notice that k is not the same thing as K)

Clustering algorithm

1. Set a positive value for *K*.
2. Initialise a matrix called weights with 1 row and m columns. Each value in this matrix should be between zero and one, and the sum of all values in weights should be equal to one.
3. Create an empty matrix called centroids.
4. Create a matrix called **S** with n rows and 1 column, initialise all of its elements to zero.
5. Select *K* **different** rows from the data matrix at random.
6. For each of the selected rows
    a. Copy its values to the matrix centroids.
    (at the end of step 6, centroids should have K rows and m columns)
7. For each row i in the data matrix
    a. Calculate the weighted Euclidean distance between data row D'_i and each row in centroids (use the weights matrix in this calculation, as per equation in the Appendix).
    b. Set S_i to be equal to the index of the row in centroids that is the nearest to the row D'_i. For instance, if the nearest row in centroids is row 3, then assign the number 3 to row i in **S**.

8. If the previous step does not change **S**, stop.
9. For each *k* = 1, 2, …, *K*
    a. Update the *k* row in centroids. Each element *j* of this row should be equal to the mean of the column D'_j but only taking into consideration those rows whose value in **S** is equal to k (i.e. those who have been assigned to cluster *k*).
10. For each v = 1, 2, …, m
    a. Update the entry v of the matrix weights (see Appendix).
11. Go to Step 7.

Euclidean distance

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

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

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:

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

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}.

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**.

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:

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