University of Essex

School of Computer Science and Electronic Engineering

CE705: Introduction to Programming in Python

Set by: Dr. Renato Cordeiro de Amorim.
Due Date: 14 January 2020 by 12:00 (UK time) via FASER.
Assignment maximum mark: 100
Proportion to final module mark: 50%

SUBMISSION REQUIREMENTS:

Students are required to submit **ONE** single .py file containing the code for this assignment no later than 12:00 (UK time) on the 14/01/2020. The standard lateness penalty will be applied to late work. Do NOT include any project file (ie. no files other than the .py).

FEEDBACK FROM THIS ASSIGNMENT

Individual feedback will be provided within 4 weeks of the due date.

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. In the text any mention to *matrix* should be read as a list of lists. Your assignment must contain the code for the functions below. If you wish you can write more functions, but the functions described below must be present. Each function **must** contain a comment **briefly** explaining why you chose to use the Python constructs (eg. control structures, lists, exceptions, internal functions etc.) the function uses.

Functions:

load_from_csv
This function should have one parameter, a file name (including, if necessary, its path). The function should read this CSV file and return a *matrix* (list of lists) in which a row in the file is a row in the *matrix*. If the file has N rows, the matrix should have N elements (each element is a list). Notice that in CSV files a comma separates columns (CSV = comma separated values). You can assume the file will contain solely numeric values (and commas, of course) with no quotes.

get_distance
This function should have two parameters, both of them lists. It should return the Manhattan distance between the two lists. For details about this distance, read the appendix.

get_max
This function should have two parameters, a matrix (list of lists) and a column number. It should look into all the elements of the data matrix in this column number, and return the highest value.

get_min
This function should have two parameters, a matrix (list of lists) and a column number. It should look into all the elements of the data matrix in this column number, and return the lowest value.

get_standardised_matrix
This function should take one parameter, a matrix (list of lists). It should return a matrix containing the standardised version of the matrix passed as a parameter. This function should somehow use the get_max and get_min functions. For details on how to standardise a matrix, read the appendix.

get_median
This function should have two parameters: a matrix (list of lists), and a column number. It should return the median of the values in the data matrix at the column number passed as a parameter. Details about median can be easily found online, eg. https://en.wikipedia.org/wiki/Median.

get_centroids
This function should have three parameters: (i) a matrix (list of lists), (iii) the list **S**, (iii) the value of K. This function should implement the Step 6 of the algorithm described in the appendix. It should return a list containing K elements, $c_1, c_2, \ldots, c_K$. Clearly, each of these elements is also a list.

get_groups
This function should have two parameters: a data matrix (list of lists) and the number of groups to be created (K). This function follows the algorithm described in the appendix. It should return a list **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.

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

It should run the algorithm (using get_groups), and show on the screen how many entities (wines) have been assigned to each group. You should run experiments with K = 2, 3, 4, 5, 6.

For example, the experiment with K=3 should run the algorithm using get_groups and then show on the screen how many entities (wines) have been assigned to group 1, how many to group 2, and how many to group 3. Clearly, if you add the number of entities assigned to each one of these groups the result should be N (the number of rows in the data matrix).

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, 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 list **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 elements.

Appendix

Data standardization

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

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

where $\bar{D}_j$ is the average of column j, $\max(D_j)$ is the highest value in column j, and $\min(D_j)$ is the lowest value in column j. D'_{ij} is the standardized version of D_{ij} – the algorithm below should **only** be applied to D'_{ij} (ie. do not apply the algorithm below to D_{ij}). Again: note that in this assignment you should represent a matrix in python as a list of lists.

Clustering algorithm

1. Set a positive value for *K*.
2. Select *K* **different** rows from the data matrix at random.
3. For each of the selected rows
   a. Copy its values to a new list, let us call it c. Each element of c is a number.

(at the end of step 3, you should have the lists $c_1, c_2, \ldots, c_K$. Each of these should have the same number of columns as the data matrix)

4. For each row i in the data matrix
   a. Calculate the Manhattan distance between data row D'_i and each of the lists $c_1, c_2, \ldots, c_K$.
   b. Assign the row D'_i to the cluster of the nearest c. For instance, if the nearest c is $c_3$then assign row i to the cluster 3 (ie. you should have a list whose i[th] entry is equal to 3, let's call this list **S**).
5. If the previous step does not change **S**, stop.
6. For each *k* = 1, 2, …, *K*
   a. Update c_k. Each element *j* of $c_k$should be equal to the median of the column D'_j but only taking into consideration those rows that have been assigned to cluster *k*.
7. Go to Step 4.

Notice that in the above *K* is not the same thing as *k*.

Manhattan distance

The Manhattan distance between the two lists **a** and **b** (both of size N) is given by

$$d = \sum_{i=1}^{N} |a_i - b_i|$$

Notice the straight brackets mean absolute value.

Marking Scheme

Characteristics of an excellent project (70% or more):

- Excellent code documentation
- Excellent use of Python's native methods and code standards
- Excellent use of relevant data types
- Follows carefully the specification provided
- Implements the described run_test, which shows the expected results.
- Excellent code optimisation in terms of memory, speed and readability
- Generally, an excellent solution, carefully worked out;

Characteristics of a good project (60%):

- Good code documentation
- Good use of Python's code standards
- Good use of relevant data types
- Follows the specification provided, with no major deviations.
- Implements the described run_test, which shows the expected results.
- Good code optimisation in terms of memory, speed and readability
- Generally a good solution, which delivers what the final user would expect.

Characteristics of a fair project (50% or less):

- No meaningful code documentation
- Code tends to be more verbose than actually needed or at times difficult to read
- No real thought on the relevance of data types
- Does not follow the specification provided (this alone will indicate a fail).
- Does not implement run_test as described, or this does not show the expected results.
- A solution that only seems to deliver what the final user would expect.

Please note:

- You must submit only one file
- You must follow the instructions for each function in terms of parameters and returns
- You should document your code.
- It is a good idea to test each function at a time, and only afterwards test the whole project