CSCI 4372/5372/6397: Data Clustering

Phase 2: Standard *K*-Means

Submission Deadline: **Wednesday, February 18 (23:59)**

Objective: Implement the standard *k*-means algorithm.

Input: Your program should be **non-interactive** (that is, the program should **not** interact with the user by asking them explicit questions) and take the following **command-line** arguments: <F> <K> <I> <T> <R>, where

- *F*: name of the data file
- *K*: number of clusters (**positive** integer greater than one)
- *I*: maximum number of iterations (**positive** integer)
- *T*: convergence threshold (**non-negative** real)
- *R*: number of runs (**positive** integer)

Hint #1: A command-line program does **not** have a graphical user interface. However, you can write (or execute) a command-line program using a graphical IDE (examples include but are **not** limited to Visual Studio, NetBeans, and Eclipse.) In other words, you do **not** need a command-line interpreter/shell to write (or execute) a command-line program.

Warning #1: Do **not** hard-code any of these parameters (*e.g.*, using a statement such as "int R = 100;") into your program (you should also **not** use meaningless variable names such as *R*). The values of these parameters should be given by the user at run-time through the command prompt. See below for an example.

The first line of *F* contains the number of points (*N*) and the dimensionality of each point (*D*). Each of the subsequent lines contains one data point in blank separated format. In your program, you should represent the attributes of each point using a **double-precision floating-point** data type (*e.g.*, "double" data type in C/C++/Java). In other words, you should **not** use an integral data type (byte, short, char, int, long, etc.) to represent an attribute. You should also **not** use a string data type (char * in C/C++, std::string in C++, or String in Java) to store the data points in memory. However, you may use non-integral data types (*e.g.*, string's) for temporary storage (*e.g.*, to store a line you read from the input file).

The initial cluster centers should be selected uniformly at **random** from the data points. A 'run' is an execution of *k*-means until it converges. A *k*-means run should be terminated when the number of iterations reaches *I* **or** the relative improvement in SSE (Sum-of-Squared Error) between two consecutive iterations drops below *T*, that is, whenever

$(SSE^{(t-1)} - SSE^{(t)}) / SSE^{(t-1)} < T$,

where $SSE^{(t)}$ denotes the SSE value at the end of iteration t (t = 1, 2, …, I) and $SSE^{(0)} = \infty$ (infinity).

Warning #2: The *k*-means algorithm does **not** involve the *square root* function (*i.e.*, sqrt(.) in C/C++/Java). If you use sqrt(.), the algorithm may **not** converge. Another function to avoid is the *exponentiation* function (*i.e.*, pow(.,.) in C/C++/Java). The *k*-means algorithm does **not** require *exponentiation* by a non-integer exponent, which is a relatively slow operation. *K*-means instead requires *squaring*, which can be performed efficiently using the multiplication operator ('*'). In short, to square a number *x*, do **not** use pow(*x*, 2); use *x* * *x* instead. In general, you should use pow(.,.) **only** if the exponent is a real (other than 0.5) or a large integer.

Warning #3: If you use a square root function **or** anything that resembles it (*e.g.*, an approximation of it) in your program, you will **automatically receive a grade of zero!**

The algorithm should be executed *R* times, each run started with a different set of randomly selected centers.

Output: Display the SSE value at the end of each iteration.

Sample Program Execution:

% F = ecoli.txt (name of data file)
% K = 8 (number of clusters)
% I = 100 (maximum number of iterations in a run)
% T = 0.000001 (convergence threshold)
% R = 3 (number of runs)
% test is the name of the executable file

```
> test ecoli.txt 8 100 0.000001 3

Run 1
-----
Iteration 1: SSE = 22.0312
Iteration 2: SSE = 18.3704
Iteration 3: SSE = 17.5163
Iteration 4: SSE = 17.3435
Iteration 5: SSE = 17.2725
Iteration 6: SSE = 17.2331
Iteration 7: SSE = 17.221
Iteration 8: SSE = 17.2185
Iteration 9: SSE = 17.2175
Iteration 10: SSE = 17.2108
Iteration 11: SSE = 17.1934
Iteration 12: SSE = 17.184
Iteration 13: SSE = 17.182
Iteration 14: SSE = 17.1766
Iteration 15: SSE = 17.1639
```

```
Iteration 16: SSE = 17.1639

Run 2
-----
Iteration 1: SSE = 18.5142
Iteration 2: SSE = 16.7108
Iteration 3: SSE = 16.07
Iteration 4: SSE = 15.6179
Iteration 5: SSE = 15.3449
Iteration 6: SSE = 15.1791
Iteration 7: SSE = 15.1201
Iteration 8: SSE = 15.0975
Iteration 9: SSE = 15.0624
Iteration 10: SSE = 15.0292
Iteration 11: SSE = 15.0162
Iteration 12: SSE = 15.0162

Run 3
-----
Iteration 1: SSE = 18.7852
Iteration 2: SSE = 16.4215
Iteration 3: SSE = 16.2141
Iteration 4: SSE = 16.1159
Iteration 5: SSE = 16.1081
Iteration 6: SSE = 16.1045
Iteration 7: SSE = 16.0995
Iteration 8: SSE = 16.0838
Iteration 9: SSE = 16.0619
Iteration 10: SSE = 16.0547
Iteration 11: SSE = 16.0327
Iteration 12: SSE = 16.0109
Iteration 13: SSE = 16.0109

Best Run: 2: SSE = 15.0162
```

Hint #2: The above is just an example. You do **not** have to run your program from the command line. However, the format of the output should be like above. Note the lack of interaction with the user, who just runs the program with appropriate parameter values.

Hint #3: Do **not** round or truncate "double" values (*e.g.*, attribute values, distances, and SSE values) in your program.

Testing: Test your program on the data sets listed in the table below (using the parameters listed in the same table). It is important to observe that in a given run, as the iterations progress, the SSE value **never** increases, that is, $SSE^{(t)} \leq SSE^{(t-1)}$ for $t = 1, 2, \ldots, I$. If you observe the exact same SSE value in two successive iterations (as in the above "sample program execution"), this

means that the algorithm has converged, and no further reduction in SSE is possible. Note that the output of your program might be different in each execution due to random initialization.

Hint #4: Here is a simple way to find out if your *k*-means implementation is **probably** correct. On the Iris Bezdek data set (K = 3), if you execute R = 100 runs of *k*-means, the possibilities include:

i. At least one run produces an SSE value less than 78.8514: Your program is **definitely** buggy. 78.8514 is the global optimum for this data set (for K = 3 clusters). It is impossible to get an SSE value less than 78.8514 unless your program is buggy.

ii. Every run produces an SSE value greater than 78.8514: Your program is **most likely** buggy. This is a small and simple data set. So, you should be able to get an SSE value approximately equal to 78.8514 in **at least** one run.

iii. None of the runs produce an SSE value less than 78.8514, and **at least** one run produces an SSE value approximately equal to 78.8514: Your program is **most likely** correct.

Data Set	***K***	***I***	***T***	***R***
Ecoli	8	100	0.001	100
Glass	6	100	0.001	100
Ionosphere	2	100	0.001	100
Iris Bezdek	3	100	0.001	100
Landsat	6	100	0.001	100
Letter Recognition	26	100	0.001	100
Segmentation	7	100	0.001	100
Vehicle	4	100	0.001	100
Wine	3	100	0.001	100
Yeast	10	100	0.001	100

Warning #4: In the "sample program execution" above, we display *only* R = 3 runs due to space considerations. As the above table shows, you **must** perform R = 100 runs on each data set.

Large Data Sets: The purpose of the large data sets (*e.g.*, Letter Recognition and Landsat) is **not** to test your patience but to motivate you to write more efficient programs. *K*-means is actually a very fast algorithm, but *only if* implemented with care (an efficient implementation can cluster millions of points within seconds on a modern PC.) If you code the algorithm in a sloppy manner (*e.g.*, by using the *pow* function rather than the multiplication operator), your program may end up being excruciatingly slow.

Programming Language: C, C++, or Java. You may **only** use the built-in facilities of these languages. In other words, you may **not** use third-party libraries.

Submission: Submit your source code (C, C++, or Java source files), output files (**for each** input file, an output file in .TXT format), and report (PDF format, see below) through Blackboard. Do **not** submit your entire project folder or the input data files I provided to you (I already have them 😊). In addition, do **not** submit your files individually; pack them into a single archive (*e.g.*, zip) and submit the archive file.

Bonus for CSCI 4372 Students [10 points]; Mandatory for CSCI 5372 and 6397 Students: The initialization approach mentioned above does **not** guard against coincident centers (this is especially problematic when dealing with data sets with duplicate points such as pixel data). If you notice a singleton cluster (that is, a cluster whose sole member is its center) at the end of any iteration, you can remedy this problem by locating the point that contributes the most to the error of its cluster and making this point to be the center of the singleton cluster. Similarly, if there are $E > 1$ singleton clusters, locate the E points that contribute the most to the errors of their respective clusters and make these points to be the centers of the singleton clusters. To test your modified program, modify the Iris Bezdek data set by artificially making 10 copies of every 5th data point. Make sure that the modified data set ("iris_bezdek_mod.txt") has 420 data points (150 original points + 150/5 * (10–1) duplicate points). Do **not** modify iris_bezdek.txt itself. Write a separate small program that reads iris_bezdek.txt and outputs iris_bezdek_mod.txt. Include iris_bezdek_mod.txt in your experiments and submission. Do **not** implement this modification unless you are certain that your basic *k*-means implementation works properly.

Bonus for CSCI 4372 and 5372 Students [10 points]; Mandatory for CSCI 6397 Students:

- [5 points] Can *k*-means initialized uniformly at random (as above) encounter empty clusters in any iteration? If yes, explain when and why this can happen. Otherwise, explain why this is **not** possible. Include your explanation as a comment at the top of your main source file.
- [5 points] Consider the case where a data point has more than one nearest center. Most implementations assign such a point to the nearest center with the smallest index. Why do you think such a scheme has become a convention in the literature? If, instead, you assign such a point to a nearest center selected uniformly at random, does the algorithm still converge? If not, why not? Add this random assignment strategy to your program as an option and explain its **theoretical** (**not** whether or not it *appears* to work in practice) convergence behavior in a comment at the top of your main source file.

Grading: *Functional correctness* and *adherence to good programming practices* are respectively worth **80%** and **10%** of your grade. Given a valid input, a functionally correct program is one that produces a correct output. There are a lot of generic or language-specific guides on good programming practices on the WWW. You **must** identify an appropriate one, include its URL at the top of your program source code(s), and use it throughout this project. In general, you should pay more attention to structural issues (*e.g.*, modularity) than cosmetic ones (*e.g.*, naming and formatting). Remember that if your program is incorrect, whether or not you adhered to good programming practices is immaterial. To earn the final **10%** of your grade, you need to write a 1−2 page report (1'' margins, single-spaced, 12-point Times New Roman font, **no** figures or tables) explaining your solution (*e.g.*, data structures and algorithms used in each step).

Hint #5: There is a discussion forum for this phase on Blackboard. As indicated in the syllabus, **frequent** and **meaningful** participation in discussion forums is part of your overall grade. You are strongly encouraged to subscribe to this forum so that you receive email notifications of posts. Remember **not** to post code fragments (or URLs pointing to code fragments) on the forum. In other words, the discussions should be about ideas and algorithms, **not** about their implementation.

Hint #6: This is **not** a program you can write in a few hours (unless you are an experienced programmer). Please start working on your program early.

Hint #7: Common problems in this phase mainly stem from the careless reading of this document. Read the document **at least** twice before working on your program. Then, read it **at least** once after you are done.