CSCI 4372/6397: Data Clustering

Phase 1: Basic K-Means

Submission Deadline: February 22 (23:59:59)

Objective: Implement the standard (batch) k-means algorithm.

Input: Your program should be non-interactive (that is, the program should not interact with the user by asking him/her 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)

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

A ‘run’ is an execution of k-means until it converges. 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.

The initial cluster centers should be selected uniformly at random from the data points. A 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 ($t = 1, 2, \ldots, I$) and $SSE(0) = \infty$.

Warning #1: The basic 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 or, even if it converges, it will converge more slowly as sqrt is a computationally expensive function (unlike +, −, x, and /, there is no direct instruction to perform sqrt in most modern CPUs). Another function to avoid is the exponentiation function (i.e., “pow” in C/C++/Java). The basic k-means algorithm does not require any exponentiation by a non-integer exponent. It only 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 the multiplication operator (x * x) instead.

Warning #2a: If you use any square root operation or 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 Output:

% 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 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
```

Testing: Test your program on the given data sets using the parameters given in the table below. It is important to observe that, as the iterations progress, the SSE values never increase, that is, SSE(t) <= SSE(t – 1) for t = 1, 2, …, I. If you observe the exact same SSE value in two successive iterations, this means that the algorithm has converged and no further reduction in SSE is possible. Note that because of random center initialization, the output of your program might be different in each execution.

Hint: Here is a simple way to find out whether or not your k-means implementation is probably correct. For 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 more 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

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 when implemented with care (an efficient C based implementation can cluster a few hundred million points within seconds on a modern PC.)

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

Submission: Submit your source code and output files (in .TXT format) via Blackboard. Do **not** submit your files individually; pack them in a single archive (e.g., zip) and submit the archive file.

Bonus for Undergraduate Students [10 points] / Mandatory for Graduate 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 an empty cluster at the end of any iteration, you can remedy this problem by locating the point that contributes the most to the overall SSE and making this point to be the center of the empty cluster. Similarly, if there are E (E > 1) empty clusters, locate the E points that contribute the most to the overall SSE and make these points to be the centers of the empty 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 small program that reads iris_bezdek.txt and outputs iris_bezdek_mod.txt. Include iris_bezdek_mod.txt in your experiments and submission.

Warning #2b: This is so important that it bears repetition: If you use any square root operation or an approximation of it in your program, you will **automatically receive a grade of zero**!