Assignment 4: k-Means Clustering

Spring 2016

100 points

Due: 11:59pm, 4/7/2016

In this assignment, you will implement a k-means clustering algorithm that can automatically determine the right value for k, the number of clusters, for a given data set which contains a set of n-dimensional data points in the Euclidean space.

The problem consists of two parts: (1) implement a regular k-means algorithm with a pre-defined k; (2) implement the method to find the right value k* for the data set, using the "elbow" method discussed in class.

Part 1: Implement k-means with a given k (50 points)

Implement a k-means algorithm which takes (a) a data set and (b) the value k as the input, and output (a) a clustering which contains k clusters of data points in the data set, and (b) the cohesion of the clustering**, measured by the average diameter of clusters in the clustering**. When a cluster has only 1 point in it, define its diameter as 0.

For the initial centroids, **pick points that are as far as possible** (i.e., use the method 1 in the lecture, refer to slide #44). Select the **first point** in the dataset as the first centroid.

<Input>

The first input argument is a .json file. Each line represents an n-dimensional data point as a JSON array. For example:

```
[2, 2]
[3.5, 4]
[0, 7]
[3.5, 2.5]
…
```

The second input argument is an integer, the desired number of clusters.

<Output> (Please print the output to standout out, stdout, do not save it to a file.)

The output should contain k+1 lines.

The first k lines are the k clusters produced by the algorithm. Each line is a list of points in that cluster, and each point is represented as an n-dimensional tuple. The order of the k lines is not important, however, for each line (cluster), the points must be sorted in ascending order (first sorted by the first dimension, then by the second dimension, and so on). For example:

```
[(0, 7)]
[(2, 2), (3.5, 2.5), (3.5, 4)]
…
```

The last line is a single float number representing the cohesion of the clustering. Please do not round the result.

<Execute>

python firstname_lastname_kmeans.py input.json k

Use the provided sample input file to test your code, set k to 3, and the expected output should be the same as provided.

<Other requirement>

To enable the reuse of the k-means algorithm you have implemented for the next task, you MUST define a function described below:

clusters, cohesion = kmeans(data_points, k)

The first input argument 'data_points' should be a list of points (as tuples), containing all the points read from the data file, e.g., [(2, 2), (3.5, 2.5), (3.5, 4), (0, 7)]. The second input argument 'k' is an integer representing number of the clusters, which can be either read from sys.argv (as in this part) or manually designated (as in the next part).

The first output 'clusters' is the result of k-means clustering, as it will not be used in the next part, you may decide its format yourself. The second output 'cohesion' is a float number representing the cohesion of the clustering.

Doing this also facilitates the graders to give credits for the next part in case your k-means implementation has some problems, so please follow the instructions above carefully.

You are strongly advised to divide your code into functional modules and comment them as detailed as possible, so that partial credits could be given, even if it fails in some test cases.

Part 2: Find k* (50 points)

There are two steps to find the number of clusters (k*) using the 'elbow' method:

Step 1. Perform m clusterings to decides the range k* falls into. In the first clustering, k = 1; in each of the subsequent clusterings, k is doubled. Stop at k = 2v when there is little change in the cohesion from k = v to k = 2v. In this assignment, we define 'little change' to be less than a given percentage (as an input argument) of change in cohesion (i.e., decrease in average diameter of clusters) comparing to the cohesion at the previous k.

Formally, stop at k = 2v when the normalized rate of change in cohesion, denoted as r, where r = |c(v) - c(2v)|/(c(v) * v), is less than a threshold θ. Note that c(k) is the cohesion of the clustering of data sets with k clusters. The threshold will be one of the input to the algorithm.

Note that it is possible than 2v is greater than the total number of points, denoted as n. In this case, just output k* = n.

Step 2. Perform a binary search for k* among the range [v/2, v]. Consider the current range [x, y], with midpoint z. If there is little change in the cohesion between z and y, then narrow the search to [x, z]. Otherwise, continue the search on [z, y]. Here to determine if the change is little, use the same normalized rate as in step 1. That is, r = |c(z) - c(y)|/(c(z) * |z-y|). If r is less than θ, we determine that the change is little.

Note that when the search is narrowed to [x, y] where y = x + 1. Output k = x or y, whichever gives the highest cohesion (i.e., lowest average diameter of clusters).

Note that the k* may turn out to be different from the ideal best number of clusters.

<Input>

The first input is a .json file the same as described in part 1.

The second input is a float in (0, 1), representing θ mentioned in Step 1.

<Output>

The output is a single integer representing k*.

<Execute>

python firstname_lastname_findkstar.py input.json theta

Use sample input file to test your code, set theta to 0.2, and the expected output should be 4.

Submission:

Submit the following 2 python scripts:

<firstname>_<lastname>_kmeans.py

<firstname>_<lastname>_findkstar.py