k-means clustering

The goal of this homework is to implement the k-means clustering algorithm we discussed in class. In particular, you will be be taking a variety of demographic data about counties in the United States and producing a clustering that shows what counties are similar to each other (in the things that this data measures).

This homework will be done using the kmeans.py file available on the class website. The file is missing several function definitions, which you must add. Below are descriptions of what each of the functions should do. You should complete the function definitions so they do the expected thing.

The website also has a file called counties.txt, which includes a variety of data on each county in the United States (including information on population, racial distribution, education levels, income levels, election results, and various other data points). You should put this file in the same folder as kmeans.py.

Exercise 1: `readData`

This function takes as input a file name, indicating the file that contains the county data. It then reads the county data in from the file. You might find it helpful to open counties.txt in a spreadsheet program like excel and in a text editor to get a feel for what it looks like. The first row contains the name of each column, indicating what type of data is available in that column, and each row after that contains the data for a particular county.

In each row, the data are delimited by semicolons. (You might find the `split()` function on strings to be helpful in breaking up the data.) The first column is the county name (including the state it is in). The second column is the FIPS code, a number social scientists use to standardize the labeling of counties across data sets. We will not be using the FIPS code here. The rest of the variables in the file we will be using, with one exception. We want to consider the nature of the county, rather than just its size, so population and land area will be misleading. Instead of using these variables directly, you should divide them to calculate the population density in the county. For each county, you should create a `County` object. (The class is defined for you.) Set the name variable equal to the name of the county, and the values variable equal to a list of the values we are considering. (Population density should replace population, and area of the county should be dropped.) Make sure the values are stored as floats, not strings.

These county objects should all be stored in a single list, in the order they are read in from the file. This list should be the returned value of the function.

For error checking purposes, the first county object in the list should have the name "`Autauga, AL`" and have the following list as its values variable:
`[25.8, 92.93789112441961, 1.2, 13.5, 51.5, 78.1, 18.4, 0.5, 1.1, 0.1, 2.7, 75.9, 3.1, 21.7, 53773.0, 11.6]`
The list of counties should have 3143 objects in it, and each object should have 16 numbers in its values variable.

Exercise 2: `normalizeCounties`

This function takes as input the list of county objects you created in the previous exercise. We now need to normalize the data so that all variables have similar spread (and therefore roughly equal weight in the following analysis). The way we will do this is to compute, for each of the 16 values, the mean and standard deviation of the value in our data. (See the Wikipedia article on standard deviation if you are unsure of the formula.) For each value in each county object, we want to replace the value with the normalized version, which is calculated by subtracting the mean of that value across the database and then dividing by its standard deviation. This function need not return any values – it should just alter the county objects it was passed as input.

Following normalization, the first county's values should have changed from those above to the following:
`[-1.1360914508849485, -0.09761783122525876, 0.25867007626415933, -0.8362612914682693, 0.6968800072452435, -0.4406592158742433, 0.6383515401822617, -0.2273153386303384, -0.09175952352172007, -0.01810789455986609, -0.45829273443586077, -0.07551781560439914, -0.5314595738793548, 0.25284846795399535, 0.6831337952039793, -0.7296777754952319]`

Exercise 3: `placeCounties`

The Cluster class has been provided for you. Each cluster has two variables, a centroid and a list of the objects it includes. You have also been provided with `initClusters`, which given a number and a list of counties creates the right number of clusters, each the first counties in the list providing the starting centroids.

The `placeCounties` function takes as input a list of counties and a list of clusters. It adds each county to the contents of whichever cluster has the closest centroid. (This is how we will update the clusters in each cycle of the k-means algorithm. We will remove all the counties from all clusters before re-adding them, so you don't need to remove a county from its current cluster before adding it to a new one.)

If the number of clusters is set to 30, then after placing the counties into the initial clusters, with no other updates, the first cluster should have 208 counties in it, and the second should have 261.

Exercise 4: `updateCentroid`

This method is in the Cluster class, but isn't provided. The centroid is a list where each element is the mean of one of the values of the county objects. (The mean should be taken over only the counties in the particular cluster.) This method takes no input and simply updates the centroid to match the set of counties currently in the cluster.

Exercise 5: `writeOutput`

The k-means algorithm is now very easy to implement, and the code for it has actually been provided for you. (All it does is repeatedly call functions that are either extremely simple or which you have written already.) What you need to add now is the function that writes the results to a file. The function writeOutput takes a list of clusters and a filename, and writes a description of those clusters to the file. You have been provided with the output file when the algorithm is run on a set of 30 clusters for 120 iterations. You should copy the formatting of that file precisely. When you run the same analysis, you should get exactly the same file as output.