[S17 HW12] files and clusters

Due April 25, 2017, 9 a.m.

[S17 HW12]

Homework 12: files and clusters

This is a two-part homework. The first part is a warm-up for the second part, asking you to write code that reads and writes files. You ought to be able to complete the two exercises for the first part during the lab meeting. They are each due next week.

The second part consists of a series of exercises for a *k*-means clustering homework that you must also complete for next week. The goal of the in-class lab is to give you practice reading and writing files before you start working on the *k*-means assignment.

Part A: file reading and writing

In each of the two exercises below, you are asked to write a function that reads an input file and writes an output file. You might refer to the example code given in the lecture slides found here (http://jimfix.github.io/math121/lab/file_rw.pdf) to help you complete the exercises.

These are each to be completed with your assigned lab partner during the lab section. The first problem should be submitted by the left partner. The second problem should be submitted by the right partner. Whatever you don't finish today should be completed and submitted by the appropriate partner for next week.

Though you do not need to use it, you might find the `split()` method on Python string objects to be useful for your coding, both for this lab exercise and for the *k*-means homework assigned today and due next week. You can read more about it in the Python documentation but below are examples of its use:

```
>>> s = "Here is an example.\n"
>>> s.split(' ')
['Here', 'is', 'an', 'example.\n']
>>> s.split('e')
['H', 'r', ' is an ', 'xampl', '.\n']
>>> t = "Here is\nAnother.\n"
>>> t.split('\n')
['Here is', 'Another.', '']
>>> t.split('z')
['Here is\nAnother.\n']
```

The method gives back a list of strings that result from dividing the given string up according to the specified "splitter" string.

Another method that's useful for processing string objects read from files is `strip()`, which gives back the string with leading and trailing whitespace (spaces, newlines, and tabs) removed.

```
    >>> "  \t\nhello \n   \t ".strip()
    'hello'
```

[S17 HW12 PA1]

Exercise 1. eligible

Write a function `def eligible(infile, outfile):` that takes two parameters, two strings `infile` and `outfile` that hold the names of two files. You can assume the first file with name `infile` already exists. It contains a list of people and their ages, separated by a comma and a space. For example, suppose `infile` was the string `"community.txt"` and that the contents of `"community.txt"` is the following:

```
Alice, 24
Bob, 13
Carlos, 19
Dave, 34
Eve, 15
```

The `eligible` function should then write out a (new) file named by the string specified as `outfile`, the second string parameter given to `eligible`. That output file's contents should be the same as the file that was input, except that people that are too young to be eligible to vote (under the age of 18) should not be included. Continuing our example, if the function call was `eligible("community.txt", "voters.txt")` then the result would be the creation of a file named `"voters.txt"` with the contents

```
Alice, 24
Carlos, 19
Dave, 34
```

You can assume that the filenames are *relative ones*,that is, that the code will be reading a file in the same folder as where the code is being run and it will write a file in that same folder.

Note that we don't provide these text files so that you can test your code. Instead, you will need to create your own test files so that you can test your code. You can use a text editor (like SublimeText) to create these files. When you create them, you should end each line with a [RETURN] (an end-of-line character that's read as `'\n'`), **including the last line**. That means that every line should follow the format *name*, *comma*, *space*, *integer*, *end-of-line*.

[S17 HW12 PA2]

Exercise 2. transpose

Write a function `def transpose(infile, outfile):` that is similar to the last exercise in that it takes two file names: a file to be read and a file to be written. In this case, the file contents consist of a header line that is a pair of integers separated by a space. The subsequent lines give a table of digit values, with a number of rows equal to the first integer, and a number of columns given by the second integer. For example, the file contents might be like this:

```
3 5
1 0 2 1 8
9 2 8 8 5
3 7 7 9 1
```

Or it might instead look something like this:

```
2 3
8 2 5
1 5 0
```

You can assume that there is at least one row and at least one column. Having read in the file, the `transpose` function should then write out a file named by the string given by `outfile` with the table "flipped" along its diagonal. For example, when reading the first file contents above it should write out

```
5 3
1 9 3
0 2 7
2 8 7
1 8 9
8 5 1
```

In the second example, it would instead output the file

```
3 2
8 1
2 5
5 0
```

Once again, you will want to create your own text files to test your code. Like in the first exercise, every line of numbers should end with a [RETURN] character, including the last. The files you output should also have this form.

Part II: 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 working from code that we provide as part of a zipped folder at this link (http://jimfix.github.io/math121/lab/hw12.zip).
When you unzip this file, you will get a `hw12` folder that includes a file named `kmeans.py` . This Python source 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 `hw12` folder 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.

Exercise 1: readData

The `readData` 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 also 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()` method on strings to be helpful in breaking up the lines of data that you read into these data components.) 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 columns 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 instead divide them to calculate the population density in the county.

For each row of county information that you read, you should create a `County` object. (A `County` class is already defined for you in `kmeans.py`.) The constructor takes two parameters, a `name` string along with a list of `values`. Make the `name` parameter equal to the name of the county (what you read in the first column of the row), and make the `values` parameter 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 `float` values, not as strings.

Your `readData` function should build a list of these `County` objects, stored in the order they are read in from the file. The function should return this list.

When you've got this code written and you run it on `counties.txt`, check that the first county object in the list has the name `Autauga, AL` and that it has the following list for its `values` attribute:

```
[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 whole list should have 3143 counties in it, and each county object should have 16 numbers in its `values` attribute.

Exercise 2: normalizeCounties

The `normalizeCounties` 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. Instead 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, 0.6383515401
822617, -0.2273153386303384, -0.09175952352172007, -0.01810789455986609, -0.45829273443586077, -0.07551781560439914, -0.5314595738793548,
 0.25284846795399535, 0.6831337952039793, -0.7296777754952319]
```

Exercise 3: placeCounties

A `Cluster` class has been provided for you. Each cluster has two attributes, a `centroid` along with the cluster's `contents` which is a list of the objects it includes. You have also been provided with `initClusters` which, when given a number and a list of counties, creates the right number of clusters. The `initClusters` function uses arbitrary starting centroids, one for each cluster, set to the values of the first several counties in the county list.

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

The `updateCentroid` method is part of 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 an output file named `output30x120.txt` that results 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.

Attempt [S17 HW12 PB5] kmeans write

Attempt [S17 HW12 PA1] eligible

Attempt [S17 HW12 PA2] transpose

Attempt [S17 HW12 PB1] kmeans readData

Attempt [S17 HW12 PB2] kmeans normalize

Attempt [S17 HW12 PB3] kmeans place

Attempt [S17 HW12 PB4] kmeans update