CS1210 Computer Science I: Foundations
Project 1: Educational Data
Part 2 due Friday, December 9 at 11:59PM

Important

CSI projects represent individual work; there are no partnerships. Everyone is responsible for their own work. Under no circumstances should you turn in the work of someone else (including any code or material you may find on the Internet) as your own, nor should you share your own work with others. You may discuss general concepts with your classmates as long as these discussions do not lead to the actual exchange of code fragments or written solutions.

You have several sources for help. Your first recourse should always be to post your question on the ICON discussion board. This is the fastest place to go for clarifications or disambiguation, or for help with Python in general (remember, don't post your solution or any part of your solution). Second, if you must share a portion of your code, you can always attend a TA help session (see the announcement on ICON entitled *General TA Office Hours start Thursday, September 1* for times and locations). You may also attend my office hours Tuesdays from 1:30 to 3:30 in my office (14 MLH). Finally, you may email your code with a specific question to me or to your TA, but please be sure that you include CS1210 on the subject line (also, this is the slowest way of getting help, due to the volume of mail we are likely to receive).

Introduction

Having read in the files for project 1, you'd like to do something constructive with these data. Inspired by our work on the presidential speeches, you decide you want to try to see if there are patterns implicit in the data that reveal underlying geographic or demographic realities. This is generally the purview of *machine learning*, and, in particular, cluster analysis. If you imagine each country being represented by a vector of values corresponding to these educational data, then each country can be mapped to a point in a multidimensional space. Countries that match to similar points would then be considered similar in terms of their underlying educational systems. If, for example, there were only two variables involved, each country would be mapped onto a plane, and points that were located near each other in the plane would then be considered similar.

Before you can use these data for clustering — or any other similarity based method — you have a lot of work to do. First, you must implicitly be able to measure distance between two points in many dimensions: but the variables you have aren't given in the same units. How can you compute distances when some variables are, *e.g.*, percentages and other variables are, *e.g.*, population counts? Second, as some of you noted previously, some of the data points correspond to real countries, while others correspond to collections of countries or other similar administrative units. You'd really like to be able to see if there are consistent patterns on a nation-to-nation comparison basis. Third, if we are to use a vector-based method (like the presidential speeches) how are we to decide which of the many variables to include in the vector measure? Finally, as our purpose becomes clearer, this proves to be an opportune time to *refactor* our code to improve its overall structure and (human) interpretability.

Code Refactoring

In this part of the project, we will restructure our code electing to adopt an object-oriented view of the system. Our code will be organized among three objects: *vectors*, *datasets*, and *analyses*. The Vector object is one we have already studied and used in our presidential speech discussion, and the code for Vector has been provided to you on ICON. The Dataset object is the new organizing object for much of the code you wrote for the first part of the project. Since we are now adopting an object-oriented

paradigm, you will need to make some changes to your existing code to integrate it into the new object (*n.b*, you may elect to start with the solution to part 1 provided rather than your own code; you will still need to make the same set of modifications). Finally, the Analysis object will be used to organize and represent an individual test of similarity — think along the lines of defining which variables are actually present in the vector itself.

Vector

The Vector object is provided in the template file on ICON. Aside from some utility methods that compute the magnitude of the vector and normalize a vector to make it a unit vector, the original Vector object contains a single method to compute the dot product (also called the scalar produce) of two vectors.

You will need to extend Vector to include a new method that returns the Euclidean distance between two vectors. The Euclidean distance between two n-dimensional vectors x and y is defined as:

$$\sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2}$$

You may use the *Vector*.*dproduct*() method as a model for how to define *Vector*.*edistance*(), which should return a single floating point value.

Of course, Euclidean distance really doesn't mean anything unless the n axes are expressed in comparable units. To see why this is so, imagine a two dimensional plot that represents height in centimeters against age in years for a sample of college students. Now consider three students: student A (180 cm, age 20), student B (150 cm, age 19) and student C (178 cm, age 23). According to the Euclidean distance formula given above, the distance between A and B is 30.02, while the distance between student A and C is about one tenth of that, or 3.6. Are these differences meaningful? More on this issue below.

Dataset

The Dataset object is where we organize most of the data we managed to read and parse in the first part of the project. At the very least, Dataset should contain four dictionaries, D, C, V and P, corresponding to the dictionaries constructed by its methods *Dataset*.*readCodes*(), *Dataset*.*readData*(), and *Dataset*.*readDefinitions*(). These methods are based on your solutions from the first part of the project, but must be modified to fit object-oriented ideas. The constructor *Dataset*.*__init__*(*self*, *codefile*, *defnfile*, *datafile*) should take three filenames corresponding to three csv files. and should invoke *Dataset*.*readCodes*(), *Dataset*.*readData*(), and *Dataset*.*readDefinitions*() appropriately.

Your existing *readDefinitions*() function can be incorporated as a Dataset method with only minimal changes that reflect the object-oriented structure. In particular, the dictionary D is no longer the constructed and returned as the value of *Dataset*.*readDefinitions*(), but rather is incorporated as a variable *Dataset*.*D* (that is, *self*.*D* within the Dataset object). *Note that the readDefinitions*() *function provided in the solution to the first part of the project correctly handles the ill-formed definition file from the World Bank discussed in class.*

The second method, *Dataset*.*readCodes*(), is a new method that reads in the specified codefile, whose first few lines look like:

```
Country Code,Country,Region
AFG,Afghanistan,West & Central Asia
ALA,Aland Islands,Europe
ALB,Albania,Europe
DZA,Algeria,North Africa
```

This is a new file (provided on ICON) that you will use to construct the dictionary of countries,

Dataset. *C*. Recall that in the first part of the project, you constructed dictionary *C* from the data file by keeping track of the country codes as you encountered them. Unfortunately, this resulted in a number of extra "countries" that weren't countries at all, like EAS (East Asia & Pacific) or ARB (Arab World). Using this new file instead will, first, allow you to focus exclusively on countries, and, second, will provide one additional piece of information, classifying each country into one of thirteen *regions*. You will also need to define a new method, *Dataset*. *readCodes*(), to read this file in and construct *Dataset*. *C* which will no longer have entries of the form:

```
Country Code : Country
```

but should rather have the new form:

```
Country Code : (Country, Region)
```

The final method, *Dataset*. *readData*() should be largely based on your existing *readData*() function (or the one provided), but with several important differences. First, *Dataset*. *readData*() need not be responsible for constructing *C* (that is now the responsibility of *Dataset*. *readCodes*()) and so it can simply ignore unrecognized country codes. Second, we no longer require *readData*() (or *makeProfiles*()) to remove or prune unusable country codes from *C*: we'll allow *Dataset*. *C* to contain all of the codes in the codefile *even if a country has no associated values*. Third, like *C*, dictionary *V* now becomes a variable within the Dataset object rather than a value returned by a function. Fourth, we incorporate the functionality similar to *makeProfiles*() in the first part of the project here by making *Dataset*. *readData*() responsible for construction *Dataset*. *P*, which now will represent a dictionary of country counts for each variable (as opposed to the first part of this project's dictionary of variable counts for each country). And, finally, we will need to postprocess the values produced so as to support the comparisons of apples to oranges, as described in the next section.

Z Scores

As noted previously, because the values in the data file are expressed using different units, it is difficult to imagine a vector space method where each axis has a different semantics. One solution is to express every variable in terms of a dimensionless value called a "Z score" or "standard score." A Z score expresses a measurement in terms of its relation to other measurements of the same value. Starting with a collection of measurements for a given variable, we can express each measurement value *v* as a function of the mean and the sample standard deviation (ssd) of the distribution of measurements observed as follows:

$$z = \frac{(v - mean)}{ssd}$$

What are the properties of these Z scores? Well, a Z score of 0 occurs when the value is exactly the mean value of the sample; a negative Z score occurs when a value is smaller than the sample mean, and a positive Z score occurs when a value is larger than the mean. If there is only one value in the sample (*e.g.*, only one country has this particular datapoint), then the corresponding Z score is necessarily 0. A larger positive (or larger negative) Z score corresponds to a value that is "more of an outlier" from the other values in the sample, assuming a normal distribution. Finally, note that these characteristics suggest a simple way of handling a missing value: just assume the missing value is 0, which is the mean of the distribution!

Computing the mean is simple; the mean is defined as the sum of the values divided by *n*, the number of values in the sample. Computing the sample standard deviation is only slightly more complicated. The ssd the square root of the variance, which can be computed as follows. First, subtract the mean you just computed from each value in the set, and square each resulting difference. Second, add these squared differences together and divide the resulting sum by $n - 1$, where *n* is the number of values in the sample.

The square root of this quotient, the variance, is the sample standard deviation. Once you have computed the mean and ssd, we can recast each of our original measurements as a Z score; see *http://www.wikihow.com/Calculate-Z-Scores* for an example.

Once you've read in the data file, your *Dataset*.*readData*() method should make a final pass over each variable in the *Dataset*.*V* dictionary and compute the appropriate Z score. Recall that *Dataset*.*V* looks something like:

```
{ 'UIS.LR.AG15T99.GPI' : {'ARE': '1.02945005893707', ... }, ... }
```

where *Dataset*.*V*.*keys*() are the names of the variables in the data file, and the *Dataset*.*V*.*values*() are themselves dictionaries indexed by country code. You will want to express all of these values as Z scores rather than the original (raw text) values from the first part of the project shown here.

Analysis

Because implementing a clustering algorithm is beyond the scope of this class, and because — if we were to truly want to perform a clustering analysis of these data — extensive Python libraries such as scikit-learning already contain implementations of common clustering algorithms, we only implement one simple similarity based search routine in our Analysis class. The method, *Analysis*.*KNN*(), when provided a country code and an integer parameter *k*, returns the names of the *k* countries most like the target country within the vector space of the Analysis. Each Analysis object represents the particular vector space used to perform a similarity search.

The Analysis class constructor takes two arguments, a Dataset object, *D*, and a vector length, *j* (5 by default). Recall from our presidential speech project that we selected the *j* most frequently occurring terms (words or word stems, ignoring stop words) with which to construct our vectors. Here, we will do the same thing, using the *Dataset*.*P* dictionary to select the *j* most common variables from which to form our unit vectors (ties in *Dataset*.*P* may be broken at random). The constructor should then create two internal dictionary values, *Analysis*.*U* and *Analysis*.*E*, where *U* is a dictionary of all the non-zero magnitude unit vectors computed from the Z scores for the *j* most common variables indexed by country code (recall countries with no data for these *j* variables will return a vector with magnitude 0, which should not be included in *U*). The *Analysis*.*E* dictionary consists of ordered lists of tuples, also indexed by country code, where each tuple is of the form (distance, other country code) and the ordering is by distance (ascending) for every country code having a vector in *U*. In other words:

```
E['ITA'][:3]
[(0.3110014043190698, 'SMR'), (0.43032244022861477, 'PLW'), (0.5582174222664916, 'AUT')
```

and, *e.g.*,

```
[(0.2854380845387539, 'CHN'), (0.37228731589824515, 'IND'), (0.5272586386843845, 'MEX')
```

Note that the Euclidean distance between two legal unit vectors will always be between 0 and 2; if one of the vectors is not legal, use None instead. One can think of *Analysis*.*E* as a two-dimensional array, indexed in each dimension by country codes.

We next turn our attention to implementing *Analysis*.*KNN*(*self*, *target*, *k*), a K-nearest neighbor search algorithm that takes two parameters, a target country code *target* and an integer count *k* (again, 5 by default) and returns a list of *k* tuples (distance, country code, country, region) representing, in decreasing order of similarity, the *k* closest countries to the specified *target*. This would be pretty simple to implement if we created E[*target*] as a dictionary with keys corresponding to other countries and values corresponding to distance from *target*. All one would have to do is to return the country codes and distances (augmented by corresponding country name and region) corresponding to the *k* smallest (''closest'') values in E[*target*]. Of course, such an implementation would not be particularly efficient. To

see why this is so, consider that each time you query an Analysis object via *Analysis*.*KNN*() for a specified target country you will need to sort a ''row'' (a dictionary) of *Analysis*.*E*. Repeated queries with the same target but presumably differing values of *k* would repeatedly resort the same dictionary. Thus a much more efficient implementation — provided this Analysis will be frequently queried — would result if we simply presorted each ''row'' *Analysis*.*E* once and for all at construction time. Then, the implementation of *Analysis*.*KNN*() would simply need to return the first *k* values in the sorted representation (again, augmented by country name and region) without having to continually resort. Thus *Analysis*.*E* should be implemented as a dictionary of tuples, where each tuple or ''row'' of *E* is just a sorted list of (distance, country code) tuples arranged in ascending order by distance as shown above.