Computer Science CS134 (Fall 2020)
Daniel Aalberts, Duane Bailey, & Molly Feldman
Laboratory 6
Data Clustering using K-Means (Due 5pm, Thursday)

Objective. Creating and using a class to organize data.

Lab overview. In this project, we will be developing a technique to *cluster* data. Clustering is a method to *classify* data (for example, muscle *vs.* neuron *vs.* liver *vs.* marrow). When data can be clustered, we can treat disparate points of data as part of a larger, cohesive whole. A clustering can also be used to identify how to classify new data values encountered in the future.

The K-Means Approach. When confronting a stream of raw, unclassified data that has some "geometric" interpretation, it is often useful to be able to partition the data into classes or *clusters* of values. If these data can be interpreted in a geometric manner, it is often useful to think of this clustering as a spatial segregation of the points.

Two functions help understand the relationships between spatial points: the notion of a *distance*, and the ability to compute the center-of-mass, or *mean* of many values. Computing the distance is important in evaluating the relative closeness of a point to several clusters. The ability to compute the center-of-mass allows us to, in a sense, label the points of the cluster with a value that is a good representation of any of them. The approach, here, is cast in spatial terms. More advanced clustering of higher dimensional data considers the notions of distance and center-of-mass more abstractly.

In our approach, we will attempt to identify exactly k clusters of data. How we select k is determined by the application. For example, we may imagine that baseballs thrown at the plate fall into a small number—4 or 5—different categories. In any case, the value k is an *input* or to the algorithm. In the end, the data will be segregated into k classes, each of which is labeled by the center-of-mass of the cluster. If the clustering is good, then the points of the cluster are represented well by their label; each point is closer to its cluster's mean than the means of other clusters.

This notion suggests a way we can generally classify new points: the distance is computed from the point to each cluster center. Each points is then assigned to the cluster whose center is nearest.

If we were given the k labels for the clusters *a priori*, the algorithm for clustering the data, D is simple:

```
Cluster data D, given k labels:
  for every point, p, in D:
    find the first cluster whose label is closest to p
    add p to that cluster
```

Notice that several cluster centers may be equally close to p. When that happens, we assign p to the first "closest cluster" encountered.

The data itself can be used to "bootstrap" informative classifications: To begin, we draw k points, at random, from the input data. Next, we go through every data point, p, and assign it to the cluster that is nearest. After all of the points have been assigned, we then recompute the centers—the means—of each newly formed cluster. We iteratively repeat the reclassification around recomputed means until points no longer move between clusters. It frequently does not require very many repetitions of this process before stability is reached.

```
Cluster data, D:
  Select k labels, L.
  repeat:
     Cluster D according to L (see above), watching for migration
  until data migration becomes sufficiently small.
```

There can be multiple stable solutions, so the k-means algorithm is typically run a few times from different initial seed points. Frequently the algorithm will converge on the same set of k means even with different initial seeds. Eventually one selects the best solution, the one that minimizes the total cluster variance (more on that later).

Application. As an example of how k-means clustering might be used, we'll investigate the compression of colors in an image. When we read an image from disk, we are able to access the individual picture elements or *pixels* as elements of a two-dimensional image array. Each pixel keeps track of its color, a value that can be encoded using the RGB color model. This model imagines that the color is generated by mixing three different lights in different levels of intensity, much as we might imaging light colors are generated on subject using red, green, and blue stage lights. If they're all turned off (they all have an intensity of zero), the stage is black. If the red light is turned full on (a high intensity), and the others are off, the stage is red. By mixing these primary light colors, we can generate new colors. For example, full intensity red, green, and blue lights will generate a white light. Thus, pixel colors are imagined as points found in a cube from color space whose dimensions are red, green, and blue intensities. To make manipulation of these values easy, the intensities are often stored as *bytes*—integers between 0 and 255, inclusive.

It is often useful to reduce the total number of colors used to represent an image. For example, if we reduced the number of colors used in Da Vinci's *Mona Lisa* with a selection of 8, we could generate a manageable paint-by-number image. If we wanted to compress an image's storage, we could replace the 24 bits needed to store a full-spectrum color with 3 bits which would identify one of 8 select colors. The 8 colors would be stored in a very small, 24 byte table saved with the image. The image would then take about $\frac{1}{8}$th of the space!

Code Review. Clone the lab resources from the gitlab repository, as usual:

```
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab06.git ~/cs134/lab06
```

As usual, we begin by carefully reading through the existing code. Your job is to fill out the code in `cluster.py`. We have also provided an application that will re-color an image using k colors, determined by clustering the image's original colors.

We have provided only a skeleton of the `Clustering` class. The methods that are present were designed to provide the functionality we expect to use in the image recoloring application. While you are responsible for designing and implementing the internals of this class, you may not change the method headers.

To help you with the design process, read through the documentation of the method headers we have provided. You might also look at the image-recoloring application to see how the `Clustering` methods will be used. You should be thinking:

- ⋆ What different types of information are stored inside the *Clustering* object? This *state* information will help you understand the attributes that will support the class.

- ⋆ The application makes method calls to *access* information inside the `Clustering`. Which types of information are *directly* stored in attributes? Is there information that can be computed on-the-fly?
- ⋆ What are the constraints on the ability of an application to modify state in the object? Obviously, when the clustering is constructed, the state is constructed, as well; what data is passed to the initializer? Does it ever change? If it does change, which methods can be identified as *mutator* methods?

With the above information in mind, review `__slots__`, the list of names of attributes that must be maintained in the structure. These attributes are initialized in the `__init__` method. Notice, by the way, that the initializer will be passed methods to compute `distance` between objects and the `mean` of several objects.

Once you have perused the files, check out the introductory video associated with this project.

Required Tasks. The focus of this week's project will be the development of the `Clustering` class. Here's what needs to be done:

1. In `__init__`, implement a better way to select the k initial labels from the data. You might consider `randint`, `choice`, or `shuffle` methods from the `random` module. Document your thinking by including appropriate comments.
2. Now, implement the `map` method. This method is responsible for determining the index of a value's ideal cluster.
3. Suppose your clustering produces k (possibly empty) labeled clusters. Complete the `relabel` method to update the label of any non-empty cluster to be mean of the values in the cluster. Labels for empty clusters can be left as is.
4. Implement `recluster`. This method is called to refine the current clustering. Keep track of (and return) the number of values that migrate to a new cluster. This information is necessary to help the algorithm identify when the clustering process has stabilized.
5. Return to `__init__` and make sure that it uses the methods you've just written to construct a stable clustering.
6. When `cluster.py` is run as a script, it generates plots of clusterings of points. Run the script a few times. You will likely observe cases where the clusters are well assigned. You may observe cases where a mean sits between two clusters and one cluster contains two means.
7. Once you are confident your class is working, try running the image recoloring application. We've included Van Gogh's `Bedroom` (a small image) and `Irises` (a large image). The result is found in idResult.png. Does it seem to work?
8. When you are finished with your implementation sign the honor code. Then, add, commit, and push your changes to `cluster.py` and `honorcode.txt`.

Extras.

1. Peruse `recolor.py`. It begins by clustering all the pixels in the image. Then, when the `image` is needed, `after` classifies each image pixel's color by mapping it to one of the k labels. Modify the recoloring code to print the k labels to the output. Make sure you submit your changes to `recolor.py` for grading.

2. Develop a *property*, `variance()`, of `Clustering` that computes the variance (the average squared Pythagorean distance) of every point from that point's cluster center:

$$\text{var} = \left(\sum_{i=0}^{k-1} \sum_{p \in \text{cluster}_i} \text{distance}(p, \text{center}_i)^2 \right) / n$$

   Observe that the variance is lower when the centers are well assigned.

3. For more credit, you can explore how the variance depends on your choice of k. Increasing k gives you more centers, so should reduce the typical distance (and, thus, distance-squared) to the centers. Experiment with a clustering and tabulate how the variance depends on the parameter k. The elbow of the resulting plot is an aid in deciding how to pick k. Add, commit, and push a plot called `elbow.pdf` that demonstrates this relationship.

⋆