Objective. Creating and using a class to organize data.

Lab overview. In this project, we will be developing a technique to cluster and classify data. 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 interpret or 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 partition data into exactly k clusters. 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. If the clustering is good, then the points of the cluster are represented well by their label; each point is, in a sense, closer to its cluster's label than the labels of other clusters.

If we are 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 labels may be equally close to p. When that happens, we assign p to the first "closest cluster" encountered.

Where do we find the labels? The data itself can be used to "bootstrap" the process. 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 whose label is nearest. Since the labels were drawn from the data, at least some of the data is well represented by the labels!

Once we have clustering, however, we have the possibility of improvement. We can compute the means of each newly formed cluster and use these as labels for a re-clustering of the data. Notice that these labels may not actually appear among the data, even though, spatially, they're
among the values they represent. When we think of the labels as centers, we can compute a variance: the sum of the squared distances from data values to their labels. The variance is small if the data are tightly clustered around their labels.

Finding a good clustering, then, involves iteratively repeating the re-clustering around recomputed means until we see no more improvement in the variance. It frequently does not require very many repetitions of this process before the variance is minimized.

Find a good k-clustering of data D:
Select k labels, L.
repeat:
Cluster D according to L (as we saw, above).
until reclustering does not reduce the variance.

There can be multiple stable solutions, so the k-means algorithm is typically run a few times from different initial clusters. Frequently the algorithm will converge on the same set of k means even with different initial seeds. From these stable solutions, one selects the most desirable clustering.

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:

```bash
git clone https://evolene.cs.williams.edu/cs134-labs/22xyz3/lab07.git ~/cs134/lab07
```

where your CS username replaces 22xyz3. As usual, we begin by carefully reading through the existing code. Your job is to fill out missing code in cluster.py. We have also provided an application, recolor.py, that will re-color an image using k colors, determined by clustering the image's original colors. There is an important change to be made there, as well.

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?

* Our intent is for the Clustering class to be immutable. We should convince ourselves that none of the methods will allow the user to modify or mutate that state of the Clustering once it has been constructed.

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. Review the __init__ method. The private attributes of the object are initialized by this method. Modify the __slots__ variable to list the four private attributes that will maintain the state of the Clustering. Notice, by the way, that the initializer will be passed methods to compute distance between objects and the mean of several objects.

When you have finished with this task, try running the linear.py script:

```
$ python3 linear.py
A 4-cluster of 12 values with variance 0.
Cluster with label 9: (9, -17, -16, -5, -3, -18, -4, 0, 1, 10, 2, 8)
Cluster with label -17: ()
Cluster with label -16: ()
Cluster with label -5: ()
```

It should describe, as we see here, a 4-clustering of 12 integers. All of the integers are found in the first cluster (cluster zero). The variance is incorrect, and multiple runs will show the cluster labels are simply drawn from the data.

2. Now, implement the classify method. This method is responsible for determining the index of a value's ideal cluster. This method should compute the distance between a particular value and each of the labels. We're looking for the index of the closest label. If there are ties, prefer the first one encountered. As you write this, realize you can use the label property. As with client applications, this property allows you to access the label tuple, but does not allow you to change it (why?).
Now, when you run `linear.py`, it should assign the data to clusters whose labels are closest:

```
$ python3 linear.py
A 4-cluster of 12 values with variance 0.
Cluster with label 9: (9,)
Cluster with label -17: (-17, -18, -16, -5)
Cluster with label 8: (8, 1, -3, 2, 0, -4)
Cluster with label 10: (10,)
```

Notice that different runs produce different labels and, thus, different clusterings. Notice how the variance—our notion of clustering tightness—is stuck at zero.

3. Now, implement the variance method. This method computes the sum of the squares of the distances from each point to its label. This will require a nested loop: the outer loop generates \( i \), a cluster index, and the inner loop sums the squares of the distances from each cluster value to its label. (Because of the loop complexity, you should avoid using list comprehensions here.) The sum returned is a measure of how tightly values are clustered.

Returning to `linear.py`, if want to make verification of the computation easier, try providing a value for \( k \) and a list of integers on the command line:

```
$ python3 linear.py 2 0 2 4
A 2-cluster of 3 values with variance 4.
Cluster with label 0: (0, 2)
Cluster with label 4: (4,)
```

You should be able to verify the variance by hand.

Running `linear.py` with our default data, one sees this:

```
$ python3 linear.py
A 4-cluster of 12 values with variance 536.
Cluster with label -5: (-5, 1, -17, 0, -3, -4, -18, -16)
Cluster with label 9: (9,)
Cluster with label 8: (8, 2)
Cluster with label 10: (10,)
```

The variance should be a value under, say, 1000. Notice that multiple runs will yield different variances. Those with smaller variance are typically better clusterings.

4. Implement recluster. The `findClustering` method develops a good clustering by iteratively reclustering data until its variance cannot be improved. The `recluster` method is called to generate a new—and possibly improved—clustering using labels that are the means of each of the current clusters. We then create and return a new `Clustering` based on these possibly improved labels. One difficulty is that our data is currently distributed throughout the current clustering. You will need to “flatten” this data, re-collecting the original data into a list by traversing the clusters.
Now, when you run linear.py as a script, it is likely to find a very tight clustering of the integers:

```
$ python3 linear.py
A 4-cluster of 12 values with variance 8.0.
Cluster with label 9.0: (9, 10, 8)
Cluster with label -17.0: (-17, -18, -16)
Cluster with label 1.0: (0, 1, 2)
Cluster with label -4.0: (-4, -3, -5)
```

Because of randomness in the process (where?) the clustering can sometimes be sub-optimal. Guaranteeing an optimal clustering can be very time consuming. Fortunately, our process seems to converge on a good clustering fairly easily.

Notice, by the way, that the `mean` method, in `linear.py`, returns 0 when it’s given an empty list. This is the label used when an empty cluster is encountered. You might think about how this might happen. When it does, it’s vitally important that we provide a reasonable guess as a label.

5. Once you are confident your class is working, move on to recolor.py. This script clusters the pixel colors of an image.

Here, you will find that the method `colorMean` computes an average color from a list of colors. Each color is represented by a 3-tuple. Currently, it assumes the color list is not empty. When an empty cluster is encountered, we expect there will be a very occasional runtime error. Modify `colorMean` to return a random color in this case. Each component of the triple should be a random integer between 0 and 255.

6. Now, try running the image recoloring application. This application will attempt to re-color an image using only $k$ colors, by clustering the pixel colors. Once a good clustering is found, the pixels are replaced by the label associated with their cluster.

Here’s how to recolor van Gogh’s *The Bedroom* using only 8 colors:

```
$ python3 recolor.py bedroom.png 8
9.54 seconds: Recolored image is in bedroom-8.png
```

The result is written to `bedroom-8.png`. Along the bottom of the image you can find the palette of colors used in the recoloring:

Van Gogh’s *The Bedroom*, left, and a recolored version, right.
Because of randomness, different runs will produce different recolorings.

7. We’ve provided four images: bedroom.png, avi.png, balloongirl.png, and irises.png. Experiment with recoloring these images with different numbers of colors.

8. At this point, if you add, commit, and push your changes to cluster.py, recolor.py and honorcode.txt, you will earn 9.5 points.

A Little Extra. If you would like to earn a little more credit, answer the following questions in the file answers.txt using the same editor you use to write python code. Make sure your answers are thoughtfully presented and that you keep your lines to under 80 characters, for ease of reading and printing.

1. The method findClustering (in cluster.py) uses randomness. Where in the method is this happening, and why is this important?

2. If we have picked the value of k poorly, the clustering is less likely to be successful. A checkerboard should probably not be colored with 5 colors. If we partition our four clusters of integers into 11 groups:

   $ python3 linear.py 11 9 -17 -18 -16 -5 8 1 -3 2 0 -4 10

   What happens?

3. A careful look at recolor.py suggests that we’re actually clustering unique pixel colors. Before they are clustered, the pixels are placed in a set. What impact does this have on performance? (Hint: Our images are roughly the same size, but some take much longer to recolor.)

   If you answer these questions, make sure you add, push, and commit your answers.txt by the deadline.

   *