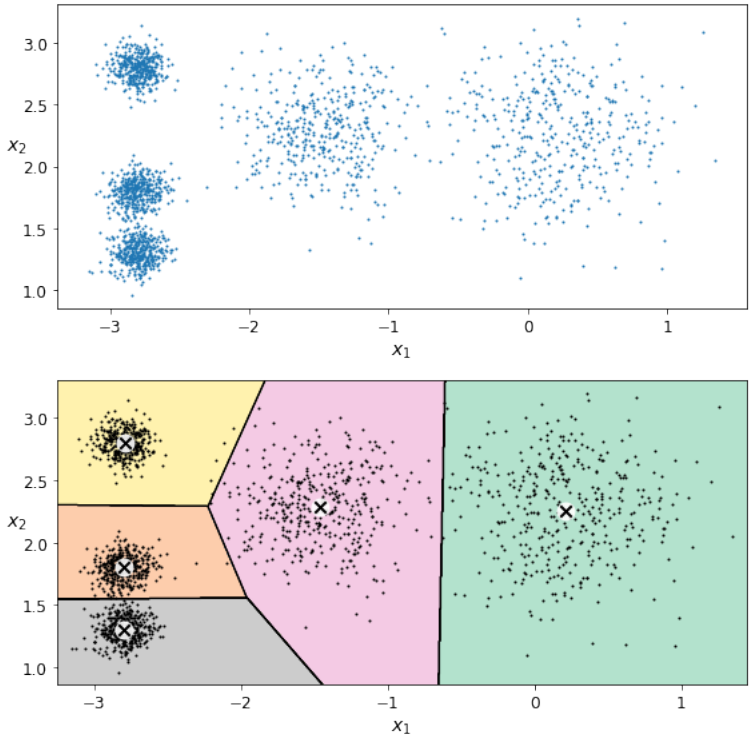
Clustering Data using the K-Means Algorithm
MATH5001: Mathematics of Machine Learning
Missouri University of Science and Technology
Instructor: Professor Yangzhi Zhang
Lab instructor: John Burkardt
19 October 2023

https://people.sc.fsu.edu/~jburkardt/classes/math5001_2023/lab/lab.pdf

---



Using KMeans, we can discover how data has formed cluster patterns.

Discovering clusters

We are interested in how much our data clusters around one or more centers.

- *The* **mean** *and* **standard deviation** *are statistical descriptions of clustering;*
- *We standardize or normalize the data to give a common scale;*
- *We use kmeans if we suspect our data forms multiple clusters;*
- *We need to choose k, the number of clusters;*
- *The* **energy** *E or "inertia" tells us how well the clusters represent the data.*
- *We can estimate a good value for k by watching how E decreases as k increases.*

1 Copying the data

Before you can carry out the following exercises, you will need to copy several datasets from the Canvas class website. If that is not available, you can try to get the information from my home page:

https://people.sc.fsu.edu/~jburkardt/classes/math5001_2023/lab/lab.html

The files we are looking for are

- *hw_data.txt*
- *faithful_data.txt*
- *ruspini_data.txt*
- *swim.jpg*

The text of this lab document is also there, as *lab.pdf*.

2 Exercise 1:

The file `hw_data.txt` contains 25,000 records, each involving three quantities, an index, a height in inches, and a weight in pounds. We want to select the first 350 records, and plot height versus weight, with the mean surrounded by three red rings of radius 1, 2 and 3 standard deviations.

Write a program `exercise1.py` and:

- use `np.loadtxt()` to read `data` from *hw_data.txt*;
- use `np.shape()` to get and print the number of rows and columns;
- print the first five rows of `data`;
- now reduce data to the first 350 rows, and last two columns of information.
- compute and print min, mean, max, variance, std of `data`;
- create `data2`, a standardized copy of `data`;
- use `plt.scatter ( )` to plot values of height (column 0) versus weight (column 1) of your standardized data;
- add 3 red rings to your plot, of radius 1, 2 and 3 standard deviations:

```
t = np.linspace ( 0, 2.0 * np.pi, 51 )
x = np.cos ( t )
y = np.sin ( t )

plt.plot (     x,     y, 'r-', linewidth = 2 )
plt.plot ( 2.0*x, 2.0*y, 'r-', linewidth = 2 )
plt.plot ( 3.0*x, 3.0*y, 'r-', linewidth = 2 )

olt.show ( )
plt.savefig ( 'exercise1.jpg' )
```

Because the data is standardized, the variance and standard deviation are both 1. The rings therefore indicate data that is no more than 1, 2, and 3 standard deviations from the mean, and should contain almost all of your data.

There is a directional bias in your data. Large heights tend to go with large weights, and small heights with small weights.

Turn in: a copy of your plot, *exercise1.jpg*.

3 Prepare for Exercise 2:

We will now read a set of two-dimensional data. We will consider regarding this data as either a single cluster, or a pair of clusters. We want to compare the energy of the one and two cluster cases.

To split our data into `k` clusters, we use the implementation of the **K Means algorithm** available in `scikit-learn`.

To use the algorithm, we need the statement

```
from sklearn import KMeans  # Capital K, Capital M!
```

If we wish to create `k` clusters, we must define the word `kmeans` as follows:

```
kmeans = KMeans ( n_clusters = k, n_init = 'auto' )
```

To cluster `data`, we then write

```
kmeans.fit ( data )
```

Following this command, we can collect some information as follows:

```
Z = kmeans.cluster_centers_
C = kmeans.labels_
E = kmeans.inertia_
```

Here, each of the `n` values `C[i]` satisfies $0 \leq C[i] < k$, and indicates that data item i belongs to cluster `C[i]`.

If we wish to plot all the points belonging to cluster 0, we might use a `scatter()` command like this:

```
plt.scatter ( data[C==0, 0], data[C==0, 1], c = 'red' )
```

with similar commands to plot points from other clusters, perhaps in blue, green, and so on.

Each cluster has a center. The coordinates of these centers are in the $k \times d$ array Z.

It's important to know how the energy changes as we increase the number of clusters. Therefore, if we want to compare the energy for the one-cluster and two-cluster cases, then for k = 1 and then k = 2, we have to:

- set `k`;
- define `kmeans`;
- apply `kmeans()` to our data;
- print `E=kmeans.inertia_`;

This was a lot of preparation, but now we are ready to deal with our tricky two-cluster data from the Old Faithful geyser.

4 Exercise 2:

Write a program `exercise2.py` and:

- use `np.loadtxt()` to read `data` from *faithful_data.txt*;
- use `np.shape()` to get and print the number of rows and columns;
- print the first five rows of `data`;
- compute and print min, max, mean, variance, std of `data`;
- create `data2`, a standardized copy of `data`;
- use `plt.scatter ( x values, y values )` to plot eruption time (column 0) versus wait (column 1);

- apply the KMeans algorithm to `data2`, requesting `k=1` clusters, and print the energy
- apply the KMeans algorithm to `data2`, requesting `k=2` clusters, and print the energy;
- in a scatter plot show cluster C=0, using `c = 'red'` and C=1 using `c = 'cyan'`;
- in the same plot, add the two cluster centers Z, using `c = 'black'` and `marker = '*'` for both

You should notice that the energy decreased a lot when we went from `k=1` to `k=2`. In your scatterplot, you should expect the red and cyan dots to be correctly clustered around a cluster center.

Turn in: a copy of your final plot, *exercise2.jpg*.

5 Exercise 3:

The Ruspini data naturally breaks into a number of clusters. We will use `KMeans` to cluster the data into $1 \leq k \leq 10$ clusters, compute the energy each time, and plot the sequence of energy values, looking for a sort of "elbow" or "hockey stick" in the plot, which suggests a natural value of `k` to choose.

Write a program `explore.py` which searches for a good value of k:

- use `np.loadtxt()` to read `data` from *ruspini_data.txt*;
- use `np.shape()` to get and print the number of rows and columns;
- print the first five rows of `data`;
- create `data2`, a standardized copy of `data`;
- use `plt.scatter ( x values, y values )` to plot x (column 0) versus y (column 1);
- for $1 \leq k \leq 10$, set up `kmeans`, cluster the data, store `E[k-1]=kmeans.inertia_`;
- print the values `k,E[k-1]`;
- plot the values `k,E[k-1]`;

Write a second program `exercise3.py` which uses your chosen value of k:

- based on your energy plot, choose a value of `k`;
- using your chosen value of `k`, define `kmeans`, use `kmeans()` to cluster `data2`;
- in a scatter plot, display each set of clustered data;
- in the same scatter plot, add the cluster centers, using `c = 'black'` and `marker = '*'`

If you have chosen `k` well, your plot should show nicely clustered data.

Turn in: a copy of your final plot, *exercise3.jpg*.

6 Prepare for Exercise 4

In this exercise, we will explore how the K-Means algorithm can be used to manipulate an image. We will look at a simple kind of compression operation in which we reduce the number of colors used. You should be familiar with the idea that in computer graphics, most images are described as an array whose entries are essentially triples of (R,G,B) (red, green, blue) values. Typically, these values are unsigned integers between 0 and 255. The arithmetic type is called *uint8*. This means that Python sees an image as a 3-dimensional array of shape $w \times h \times d$ of uint8 values, where d is 3.

A jpeg image stored as a $1200 \times 900 \times 3$ *array of uint8.*

We would like to regard the colors as data to be clustered. In particular, we want to create K color clusters, such that each color used in the image is assigned to one of these clusters.

Each cluster can be represented by its center value Z.

We reason that if each color is close to its cluster center Z, we might be able to simplify the picture so that only K colors are used, namely, the K cluster centers Z.

We can try to create a new image, using just these K colors.

This is a standard process in image compression and image analysis.

Because the image is stored as a triple index array of unsigned integers, we need to do some mysterious manipulations to convert the image to a numpy array, process it with kmeans(), and then convert the result back into something that looks like an image that we can display!

7 Exercise 4:

To begin this exercise, we will import KMeans, matplotlib.pyplot, and numpy as usual, but we will also need a library that can read and write image files:

```
import imageio.v2 as imageio
```

Assuming we have a copy of the graphics file, we now read it into the program, and immediately display:

```
image = imageio.imread ( filename )
plt.imshow ( image )
plt.show ( )
```

The image currently is represented by unsigned 8 bit integers. We want to convert this to floating point numbers between 0 and 1:

```
image = np.array ( image, dtype = np.float64 ) / 255
```

The `kmeans()` code expects to work on an array with two dimensions, but our image data is in 3D. We need to reshape the array so that all the pixel data is in the first dimension. First we retrieve the current dimensions of the 3D array (how do we ask the array to tell us its shape?). Then we reshape the array so it looks like a 2-dimensional numpy array that kmeans() can handle:

```
image = np.reshape ( image, ( w * h, d ) )
```

Now it's time for you to remember three commands that you need, which specify a value for k, the number of clusters, create a K-Means instance, and use it to fit the data in `image`.

What goes in the following lines?

```
k = ?
kmeans = ?
kmeans.fit ( ? )
```

The call to fit the data will automatically create a number of attributes, of which we will need

- **C** is the index of the cluster each color belongs to;
- **Z** is the center of a cluster, an "average" color;

so we give these short names as follows:

```
C = kmeans.labels_
Z = kmeans.cluster_centers_
```

Now comes the somewhat magic step. We create a new array, in which the original colors are replaced by the cluster centers. Think about what is happening in the following command:

```
image2 = Z[C]                          # Replace every color by its "center"
image2 = image2.reshape ( w, h, d ) # Restore original image shape
```

You can peek at the result using the `imshow()` and `show()` commands.

Now we want to convert our image back to unsigned integer arithmetic:

```
image2 = np.array ( 255 * image2, dtype = np.uint8 )
```

and we can save the result

```
imageio.imwrite ( 'exercise4.jpg', image2 )
```

If you experiment with the value of k, you can see that although the original photo has the choice of 256^3 distinct colors, you can make a perfectly acceptable image with much fewer than 50!

Turn in: a copy of your plot, *exercise4.jpg*.