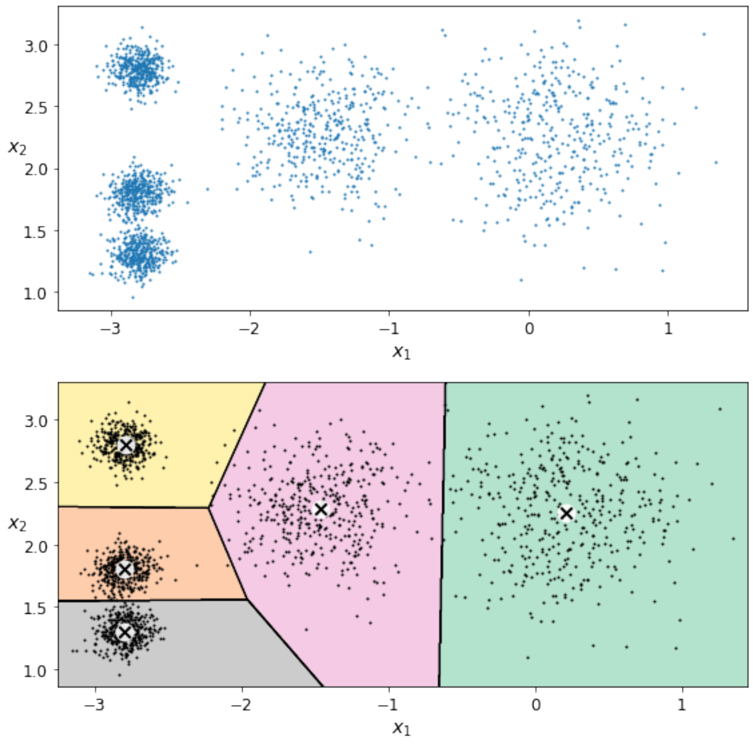
Clustering Datasets using K-Means
Mathematical Programming with Python

MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/∼jburkardt/classes/python_2025/cluster2/cluster2.pdf



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

1 Dataset 1: A single cluster

The file `hw_data.txt` contains 350 records, containing student height in inches, and weight in pounds. We want to plot height versus weight, with the mean surrounded by three red rings of radius 1, 2 and 3 standard deviations.

```
  filename = 'hw_data.txt'

  data = np.loadtxt ( filename )
  rows, cols = np.shape ( data )
#
#  Statistics.
#
  print ( '    np.min(data,axis=0):   ', np.min ( data, axis = 0 ) )
  print ( '    np.mean(data,axis=0):  ', np.mean ( data, axis = 0 ) )
  print ( '    np.max(data,axis=0):   ', np.max ( data, axis = 0 ) )
  print ( '    np.std(data,axis=0):   ', np.std ( data, axis = 0 ) )
  print ( '    np.var(data,axis=0):   ', np.var ( data, axis = 0 ) )
#
#  Standardize.
#
  data = ( data - np.mean ( data, axis = 0 ) ) / np.std ( data, axis = 0 )
#
#  Scatter plot.
#
  plt.scatter ( data[:,0], data[:,1] )
#
#  Three red circles.
#
  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 )
  plt.axis ( 'equal' )
  plt.show ( )
```

Because the data is standardized, the variance and standard deviation are both 1, and the data tends to cluster in a circle. The red rings therefore indicate data that is no more than 1, 2, and 3 standard deviations from the mean, and should contain almost all the data.

You can see a directional bias in the data. Large heights tend to go with large weights, and small heights with small weights.

2 Dataset 2: Automatic processing of Old Faithful

Our second dataset is the Old Faithful information about the eruption wait and length. We use a `scipy()` function `kmeans2()` to group the data and examine a plot of the results.

```
  from scipy.cluster.vq import kmeans2

  k = 2
  Z, C = kmeans2 ( data, k )
```

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

```
  Z = array of cluster centers, one per cluster
  C = cluster assignment, one per each data item
```

If we wish to plot all the points belonging to cluster 0, we 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.

It's important to know how the energy changes as we increase the number of clusters. Since `kmeans2()` doesn't report the energy directly, we have to compute it ourselves.

Here's part of the program that makes 2 clusters:

```
  from scipy.cluster.vq import kmeans2
  import matplotlib.pyplot as plt
  import numpy as np

  data = np.loadtxt ( 'faithful_data.txt' )

```

```
data = ( data - np.mean ( data, axis = 0 ) ) / np.std ( data, axis = 0 )

k = 2
Z, C = kmeans2 ( data, k )

plt.scatter ( data[C==0,0], data[C==0,1], c = 'red' )
plt.scatter ( data[C==1,0], data[C==1,1], c = 'cyan' )
plt.scatter ( Z[:,0], Z[:,1], c = 'black', s = 250, marker = '*' )
plt.axis ( 'equal' )
plt.show ( )
```

and here's one way to compute the energy for a given clustering:

```
d0 = ( data[:,0] - Z[0,0] )**2 + ( data[:,1] - Z[0,1] )**2
d1 = ( data[:,0] - Z[1,0] )**2 + ( data[:,1] - Z[1,1] )**2
E0 = sum ( d0[C==0] )
E1 = sum ( d1[C==1] )
E = E0 + E1
```

3 Dataset 3: Searching for a good value for k

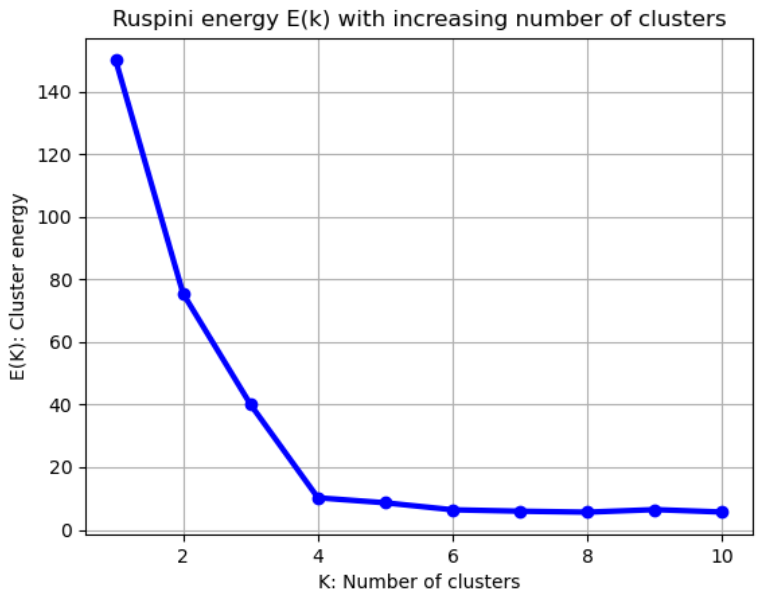
The data in `ruspini_data.txt` naturally breaks into a number of clusters. We will use `KMeans` to cluster the data into $1 \leq k \leq 10$ clusters and plot the sequence of energies, looking for a sort of "elbow" or "hockey stick" in the plot, which suggests a natural value of `k` to choose.

We begin by reading the data, standardizing, and plotting it. This already gives us a guess for what a good value of k will be.

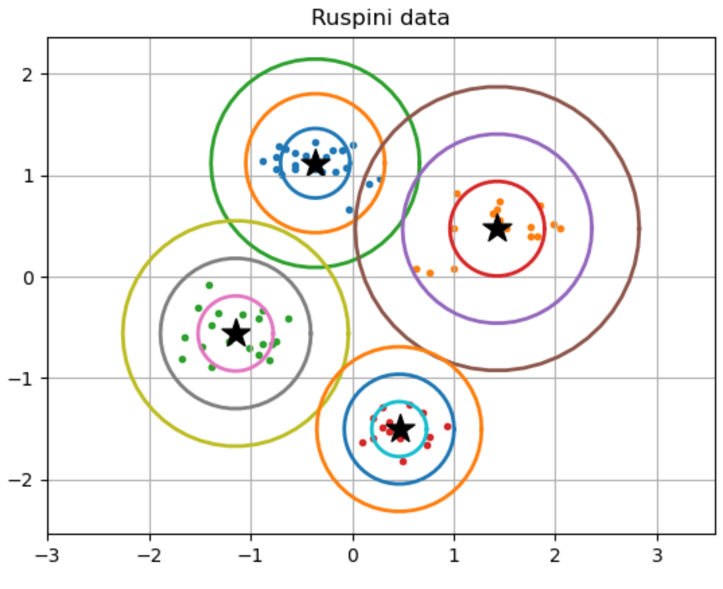
Here is how we can systematically try various clusterings, and report the energy.

```
kmax = 10
E = np.zeros ( kmax )
for k in range ( 1, kmax + 1 ):
  Z, C = kmeans2 ( data, k )
  for i in range ( 0, k ):
    bd = ( data[:,0] - Z[i,0] )**2 + ( data[:,1] - Z[i,1] )**2
    E[k-1] = E[k-1] + np.sum ( bd[C==i] )
  print ( '  %d  %g' % ( k, E[k-1] ) )
```

When we plot the energies, we see a clear message;



Now we can go back and use the indicated value of k to get a good clustering of our data:



4 Dataset 4: Reducing Colors in a JPEG Image

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.

Our target will be a JPEG image. You can imagine a digital color image as a 3-dimensional array, involving a pixel grid that is a $w \times h$ grid of pixels, with each pixel being assigned R, G and B intensities that define a color. Typically, each color intensity might be an 8-bit integer between 0 and 255, allowing for more than 16 million distinct color choices.

What makes everything harder is that, because digital images can be so large, they are typically stored in a special arithmetic format sometimes called *uint8* for "unsigned 8 bit integer" and various compression methods have been applied to squeeze the file down in size. Thus, before a JPEG file makes sense to us, we have to translated it back to something that looks like a `numpy()` array.

Our clustering method will try to find just a few clusters, such that every pixel in the image has a cluster color that is close. Since the `kmeans2()` algorithm expects our data to be a 2D array of `n` examples of `d` dimensional things, we will have to temporarily reshape our image to have `w*h` rows, each containing a list of 3 values, the RGB information. After we have done the clustering, we can reshape our image and then restore it to looking like a JPEG again.

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

Converting from `JPEG` to `numpy` array and back will be handled by a package called `imageio`.

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

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

```
filename = 'swim.jpg'
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:

```
data = 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 listed in a 2D table instead. First we retrieve the current dimensions of the 3D array. Then we reshape the array so it looks like a 2-dimensional numpy array that kmeans() can handle:

```
data = np.reshape ( data, ( 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 = 10
Z, C = kmeans2 ( data, k )
```

- **Z** is a set of "average" colors;
- **C** is the average color nearest to each pixel's color;

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

```
data2 = Z[C]                        # Replace every color by its "average"
data2 = data2.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 * data2, dtype = np.uint8 )
```

and we can save the result

```
imageio.imwrite ( 'exercise4.png', 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!

If I check the size of the original image and the 10 color image, I see a drastic reduction in size:

```
-rw-r--r-- 1 john john 632994 Sep 30  2023 swim.jpg
-rw-r--r-- 1 john john 269803 Mar 27 19:45 swim_10_colors.jpg
```