Clustering Old Faithful Data 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/cluster/cluster.pdf

The Old Faithful geyser isn't quite as regular as imagined.

1 Clustering for data analysis

Clustering begins with an unorganized collection of data, and separates the data into groups, based on some measure of similarity. An algorithm is used to decide the number of these groups and how they are to be defined. A quantity called the "energy" or "total variance" can be used to evaluate the quality of a given clustering.

A simple example of clustering involves a set of points scattered in the plane. We may use distance as the measure of similarity, and create groups of points that are clumping together.

The goal of clustering is to search for patterns revealed by the way the data seems to organize itself.

Clustering can be considered as a machine learning algorithm. It fits most properly in the subcategory of "data analysis". It is considered an unsupervised classification procedure.

2 Examples: 100 points in the plane

To illustrate the task of clustering, we begin with some artificial sets of $n = 100$ data points, whose typical entry is (x_i, y_i):

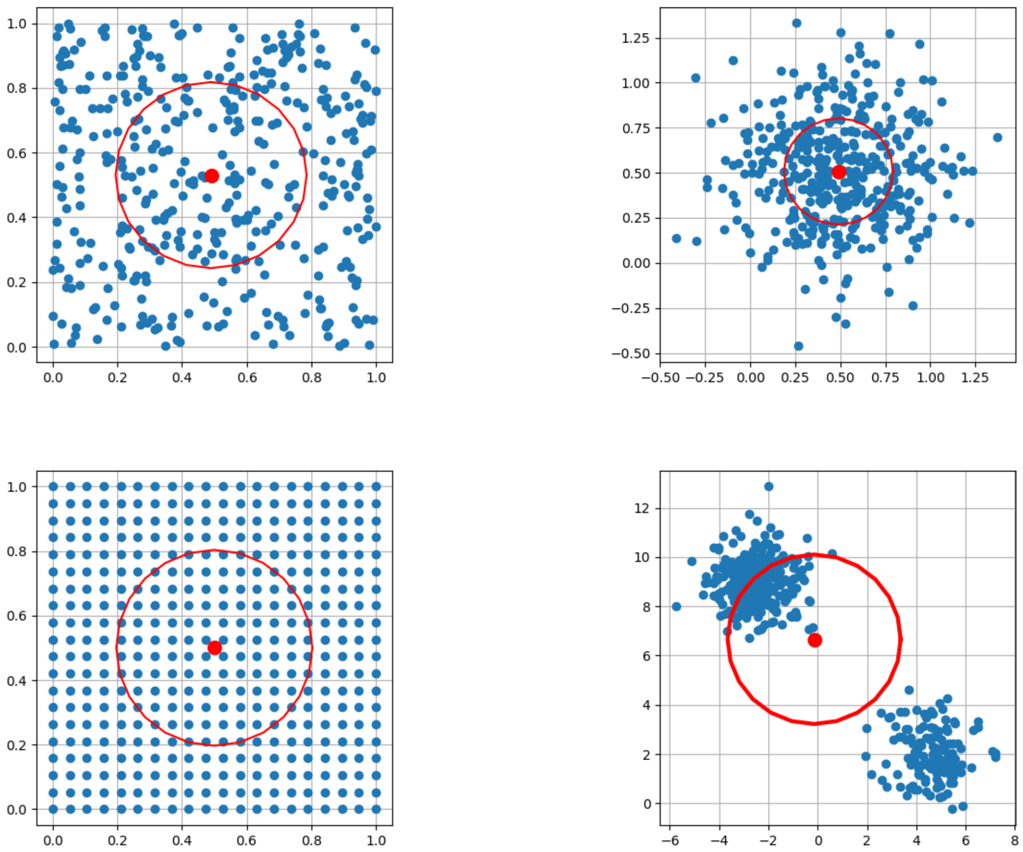
- **U**: uniform random values in the unit square;
- **N**: normal random values with mean $\mu = 0$ and standard deviation $\sigma = 1$;
- **G**: a 10×10 grid of points in the unit square;
- **B**: two "blobs", with one blob having twice as many points as the other.

If we store our data in an array Z of shape $(2, n)$, we can compute some useful statistical measurements, each of which will be returned as a pair of values, for the x and y components:

- `np.min(Z)` : the minimum value;
- `np.mean(Z)`: the mean or average value;
- `np.max(Z)` : the maximum value;
- `np.var(Z)` : the variance or σ^2;
- `np.std(Z)` : the standard deviation σ.

These numbers give us a little idea of the behavior of the data, that is, the box in which it lies, the center of the data, and the size of a "fence" we can draw to capture many of the data values.

Since our example data is two-dimensional, and not too numerous, we can simply plot the values to see what is going on:



The uniform random data exhibits no pattern. The normal random data does seem to be well described by its mean value, and the standard deviation. Our "fence" of radius one standard deviation captures a large amount of the data. The grid data certainly has a pattern. However, the statistical measurements are almost the same as for the uniform random data, so they don't tell us anything new. In contrast, the blob data is clearly patterned, but the statistical data actually seems to have a very poor understanding of what is going on. The blob data could be understood if we realize that there are two separate clusters, with two separate means and standard deviations.

Many machine learning datasets contain clusters like this, in which thousands of data items, each involving tens of measurements, naturally fall into several different groups. If we can distinguish the location and shape of these clusters, we may be able to do a much better job of understanding our data.

The K-Means algorithm was developed to automatically search for a good division of data into clusters. We will show how this algorithm is used, and how, to get the best results, the user must still make some decisions and adjustments, including scaling the data, choosing the number of clusters to seek, and modifying the similarity measurement.

3 The Geyser Data

Geysers are natural fountains of water which repeatedly erupt and then pause. A famous example in Yellowstone Park is known as "Old Faithful", which at one time seemed to erupt regularly, about once an hour.

We have received a file *faithful_data.txt* containing 272 pairs of measurements of the duration of a single eruption, and the subsequent wait or "quiet time" that follows.

We read the data with the command

```
data = np.readtxt ( 'faithful_data.txt' )
x = data[:,0]
y = data[:,1]
```

Our goal is to search for any patterns or regularity in the data.

3.1 The Wait Times

We begin by focusing on the wait times in y, and calculate statistics:

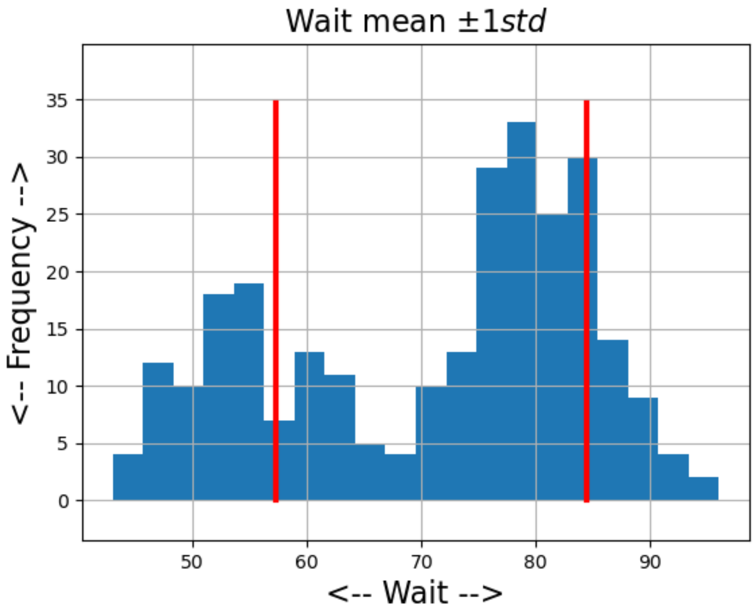
```
  y.shape =  (272,)
  np.min(y)    =  43.0
  np.mean(y)   =  70.8970588235294
  np.max(y)    =  96.0
  np.var(y)    =  184.14381487889273
  np.std(y)    =  13.569960017586371
```

Now we might imagine that a plot of the wait data time would be some kind of bell curve, centered at 70, with the majority of the values within one standard deviation, that is, roughly in the interval $[58, 84]$.

To verify this, we can request a histogram

```
  plt.hist ( y, bins = 20 )
```

but the results are not quite what we expect:



The data is not a bell curve, and seems actually to mostly avoid the mean value of 70, and instead seems to have a double hump.

3.2 Considering Duration and Wait Together

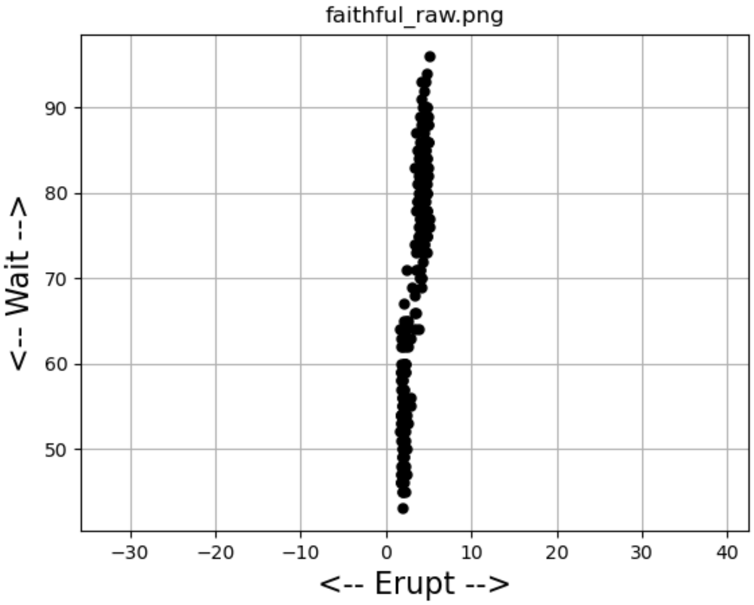
To get a better feel for what is going on, let's consider the bigger picture, that is, the pairs (x, y). First we compute statistics:

```
data.shape =  (272, 2)
np.min(data)   =  [ 1.6  43. ]
np.mean(data)  =  [ 3.48778309  70.89705882]
np.max(data)   =  [ 5.1  96. ]
np.var(data)   =  [   1.29793889  184.14381488]
np.std(data)   =  [ 1.13927121  13.56996002]
```

We plot these as points on a 2D picture. Notice that the x values are much smaller than the y values. The graphics software wants to make the plot "readable",. and so will stretch the x scale to make a roughly square picture. But this hides what is going on. To use the same scale for both quantities, type:

```
plt.plot ( x, y, 'k.', markersize = 10 )
plt.axis ( 'equal' )
```

and this is the awful, but correct result:



We are planning to examine the data in terms of which points are near each other. But when it is strung out like this almost in a one-dimensional line, our model will be very poor. The problem is that the x and y components have very different scales. To do the kind of modeling we plan, we need to have the two components rescaled.

3.3 Rescaling Data

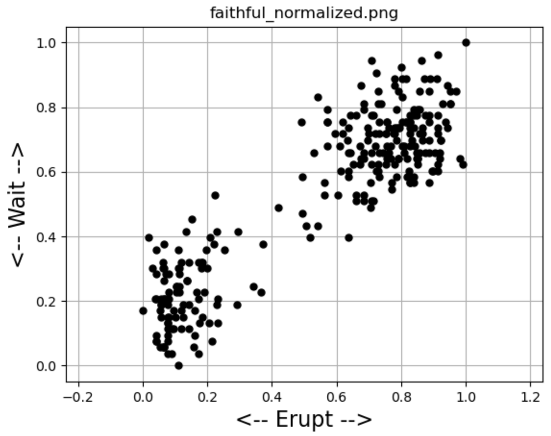
If we *normalize* the data, we will use a linear mapping which preserves the shape of the information, but squeezes or stretches each component separately, to fit into the $[0, 1]$ interval as follows:

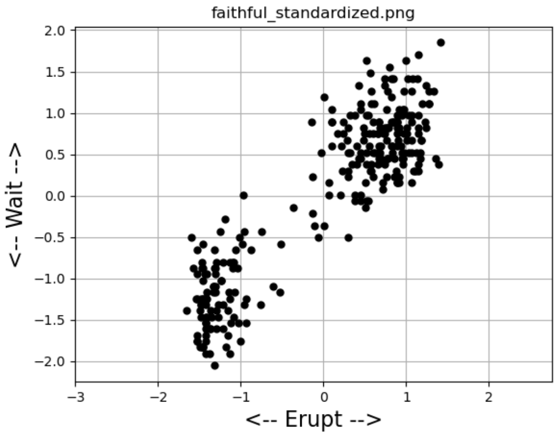
$$
\begin{aligned}
xn &= \frac{x - \min(x)}{\max(x) - \min(x)} \quad \text{Normalization} \\
yn &= \frac{y - \min(y)}{\max(y) - \min(y)}
\end{aligned}
$$

An alternative is *standardization*, in which we center the data around the mean, and give it a standard deviation σ of 1:

$$
\begin{aligned}
xs &= \frac{x - \bar{x}}{\sigma(x)} \quad \text{Standardization} \\
ys &= \frac{y - \bar{y}}{\sigma(y)}
\end{aligned}
$$

When we plot the normalized and standardized data, using the same scale on the x and y axes, we see that the data is nicely spread out in both components:





The normalized data is forced to stay within the unit square, while the standardized data is centered at 0, and most, but not all, of the data is within one standard deviation of the center.

If you simply plot the original data without forcing equal scaling, you will get a similar picture to what we see here. However, if we use the original bad scaling, our clustering results will fail badly.

From now on, we will work with the normalized data in the unit square, after creating a new version of the data in the file *faithful_normalized.txt*:

```
xn = ( x - np.min ( x ) ) / ( np.max ( x ) - np.min ( x ) )
yn = ( y - np.min ( y ) ) / ( np.max ( y ) - np.min ( y ) )
datan = np.column_stack ( [ xn, yn ] )
np.savetxt ( 'faithful_normalized.txt', datan )
```

3.4 A Single Cluster

Let's start by thinking of describing our data as a single cluster.

It is natural to describe a cluster by Z, the center it clusters around, and E, a measure of how tightly the data stays near the center, often called the "energy" of the cluster.

If we have chosen a center point Z, then E is the sum of the squared distances from each data point to Z:

```
E = np.sum ( ( data[:,0] - Z[0] )**2 + ( data[:,1] - Z[1] )**2 )
```

For the case of a single cluster, the energy E can be converted to the statistical variance if we divide by the number of data items n.

It turns out that there is a natural choice for Z, because of the following theorem:

Theorem 1 *Of all possible values of the cluster center Z, the mean of data minimizes the energy E.*

Therefore, we plan to set the cluster center with the command:

```
Z = np.mean ( data )
```

3.5 Guessing a Pair of Clusters

We have seen that the geyser data is not well described as a single cluster. It looks much more natural to describe it in terms of two clusters. Can we generalize our approach to handle this case?

We could do so by:

- identifying two separate cluster centers or “means” Z_0 and Z_1;
- assigning data item i to the nearer of the two cluster centers, so that data item i belongs to cluster C_i (a value of 0 or 1);
- computing separate energies e_0 and e_1, so $E = e_0 + e_1$.

For the plot of our geyser data, it’s easy to estimate two centers Z:

- Center Z0: (0.05,0.2)
- Center Z1: (0.8,0.75)

We assign data item (x_i, y_i) to the nearest center, that is, for each `i`, we compute

```
Z = np.array ( [
  [ 0.05, 0.20 ],
  [ 0.80, 0.75 ] ] )

d0 = np.linalg.norm ( data[i,:] - Z[0,:] )
d1 = np.linalg.norm ( data[i,:] - Z[1,:] )
```

and then assign the data item to cluster `y[i]` where

$$C_i = \begin{cases} 0, & \text{if } d_0 \leq d_1 \\ 1, & \text{if } d_1 < d_0 \end{cases}$$

which we can do by:

```
C = np.zeros ( n, dtype = int )

c0 = np.where ( d[:,0] <= d[:,1] )
n0 = np.sum   ( d[:,0] <= d[:,1] )
C[c0] = 0

c1 = np.where ( d[:,1] <  d[:,0] )
n1 = np.sum   ( d[:,1] <  d[:,0] )
C[c1] = 1
```

And we can compute the energy of each cluster:

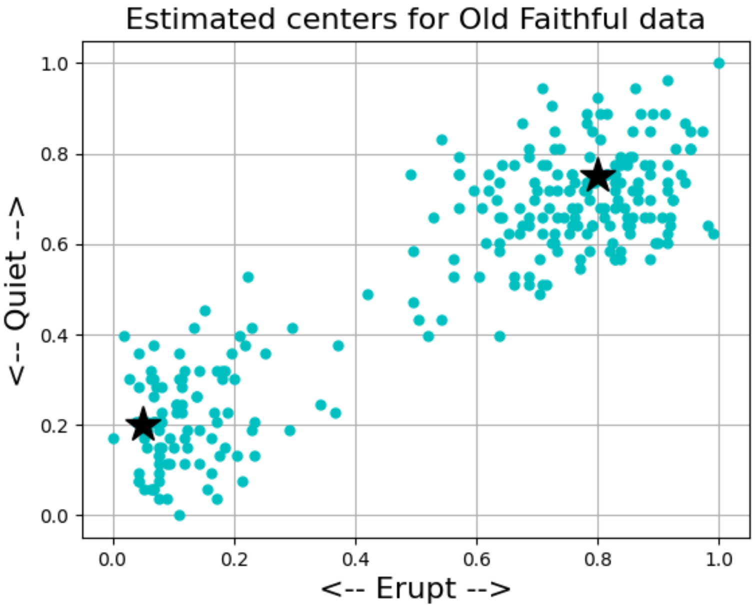
$$\begin{aligned} e_0 &= \sum_{C[i]=0} (data[i,0] - Z[0,0])^2 + (data[i,1] - Z[0,1])^2 \\ e_1 &= \sum_{C[i]=1} (data[i,0] - Z[1,0])^2 + (data[i,1] - Z[1,1])^2 \\ E &= e_0 + e_1 \end{aligned}$$

or, in Python,

```
e0 = np.sum ( ( data[C==0,0] - Z[0,0] )**2 + ( data[C==0,1] - Z[0,1] )**2 )
e1 = np.sum ( ( data[C==1,0] - Z[1,0] )**2 + ( data[C==1,1] - Z[1,1] )**2 )
E = e0 + e1
```

Notice that, unless we did a terrible job picking the two centers, the new value of E should be less than it was for a single cluster. This is because in the one cluster case, we are summing the distance from every data item to a single center, but in the two cluster case, there are two centers, and each data item is compared to the nearer one. To exaggerate, it’s the difference in total mileage (squared) if everyone has to drive to Washington DC, or everyone has to drive to their state capital.

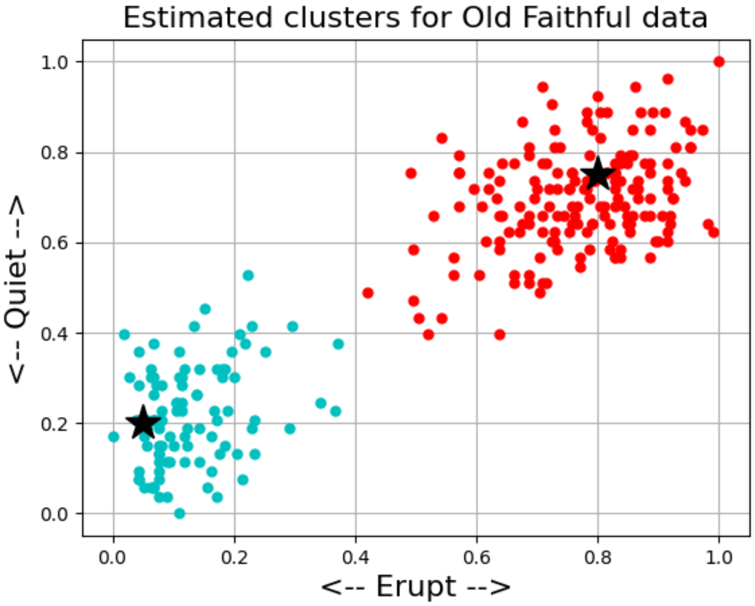
Let’s go ahead and see how our guess at cluster centers will work!



Old Faithful Data, with rough guess at two cluster centers

We can display the clustering using commands like

```
plt.plot ( data[C==0,0], data[C==0,1], 'c.', markersize = 10 )
plt.plot ( data[C==1,0], data[C==1,1], 'r.', markersize = 10 )
```



Old Faithful Data, after clustering data to our two guessed centers

Compare the one cluster and two cluster energies:

```
  1 Cluster size  272               energy E = 0.172
  2 Cluster size  272 = 97 + 175   energy E = 0.053 = 0.023 + 0.030
```

So, even with just an obviously poor guess for the center locations, we did a pretty good job of reducing the cluster energy.

However, this job was pretty easy because there was only a small amount of data, it was only two-dimensional, and it only had to be broken into two clusters.

We clearly didn't reach the optimal arrangment, since the "centers" are not at the centers of the clusters. But if we move them, then the cluster assignments might change too. Can we keep going back and forth, adjusting centers and clusters?

And what happens with bigger data sets, higher dimensions that we can't plot, and data that naturally breaks into more clusters?

We need to find a general algorithm that can automatically make better choices than we did, and on bigger and harder problems.

4 The K Means algorithm

The K-Means algorithm is designed to search for the best arrangement of n data items x of d dimensions into k clusters. That means it should produce

- Z: a set of k centers, each of dimension d;
- C: a vector of length n, assigning data x_i to cluster $C(i)$;
- e: a vector of length k, the energies of each cluster, or just E, the total energy.

We seek a clustering which minimizes the total energy E, that is, the sum of the individual e vectors,

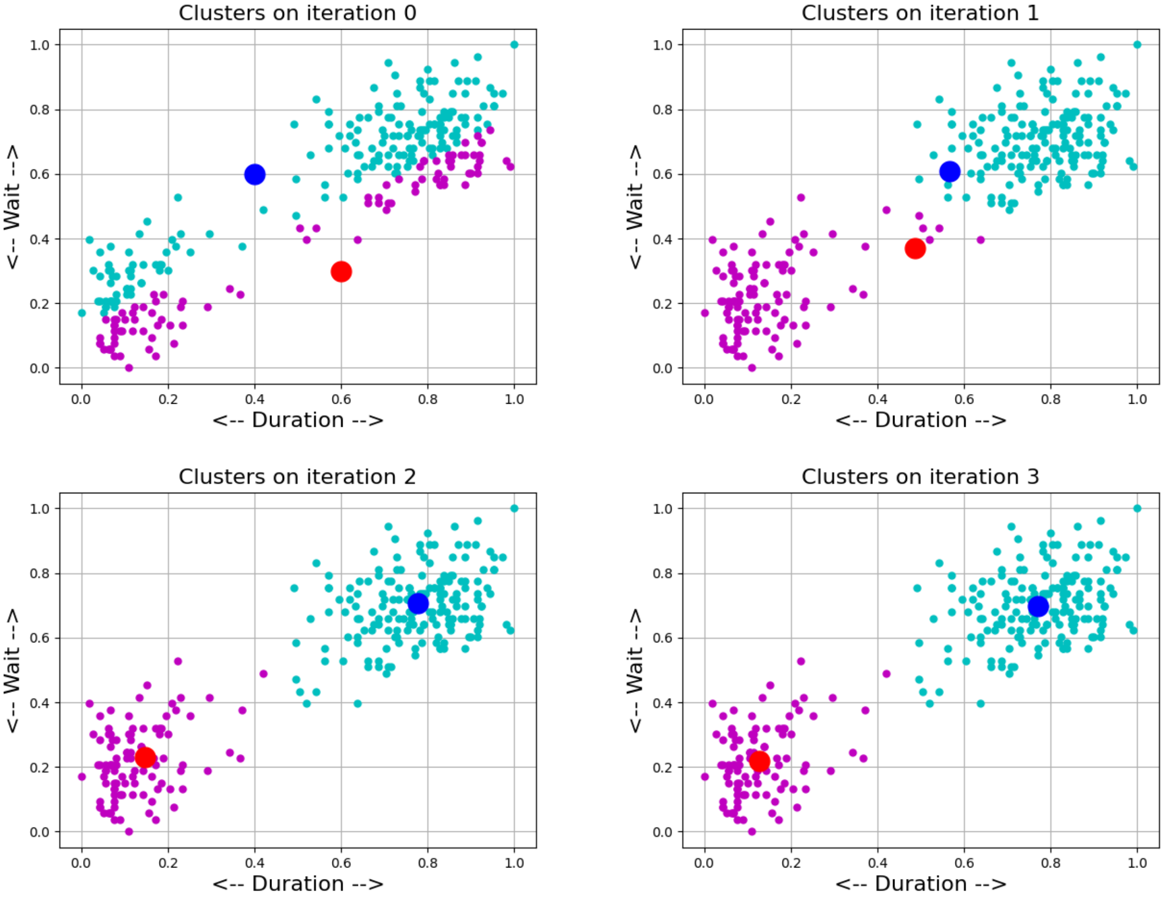
We have already tried an example of these steps by hand, picking some centers and then assigning each data item to the nearest one. Let's think of our example as a suggesting the first step of an iteration. How should we proceed?

1. Initialize the centers Z in some way;
2. Set the cluster $C(i)$ for data i as the nearest center Z;
3. Replace each Z by the mean of the data assigned to it;
4. If things have changed and we are allowed more iterations, return to step 2;
5. Compute total energy E and stop.

Luckily, the adjustments that need to be made become smaller and smaller, and at some point, the change in the location of the centers is so small that no data item will switch its cluster assignment. At that point, the iteration is done, and the clustering has been computed.

For very large datasets, it may be necessary to halt the iteration before this perfect result is achieved, by specifying a tolerance for the motion of the centers, or some other test that means we may stop a little early.

Here are the first four steps in a simple user-written version of the K-Means iteration:



Four iterations of the K-Means algorithm

Even with a terrible set of starting centers, we can see that the iteration steadily improves. Moreover, it should be clear that the same algorithm can be applied to sets of data that are large or high dimensional or for which the number of clusters is thought to be big.

The K-Means algorithm is important enough that several versions are built into the `scipy()` library.

5 `kmeans2()` computes a pair of clusters

We managed to create two reasonable clusters for the Old Faithful data by plotting the points and estimating two centers. We would prefer not to have to make such eyeball estimates, and we certainly can't do this for data in high dimensions. Let's see how we would go about clustering our data using the `scipy` implementation of the K-means algorithm.

We import a version of the K-Means algorithm as follows:

```
from scipy.cluster.vq import kmeans2
```

We read `data`, a normalized version of our data from the file *faithful_normalized.txt*. For convenience, we make separate copys `x, y` of the normalized x and y components of our data. Then we display a scatterplot to see what we are dealing with.

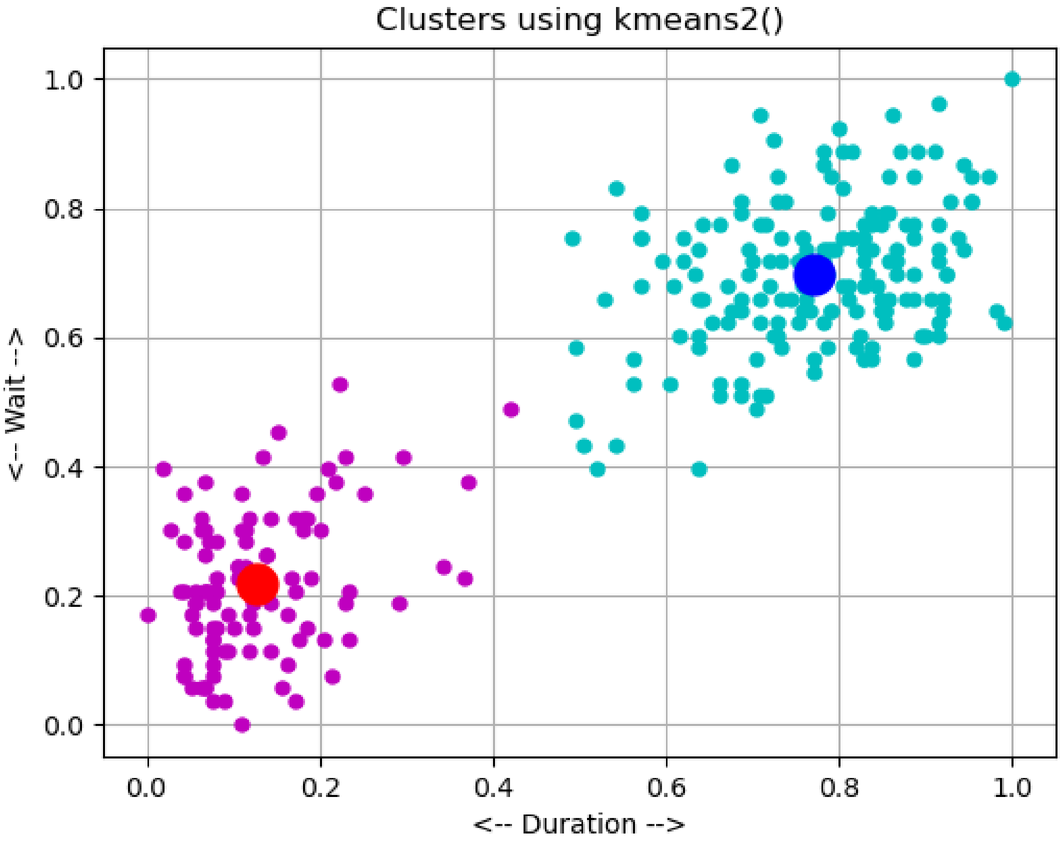
Now we choose `k`, the number of clusters to request, and call `kmeans2`, which returns 2 cluster centers and a data label (0 or 1) for each data point.

```
#
#  Choose the number of clusters, and call kmeans2().
#
  k = 2
  c, label = kmeans2 ( datan, k )
#
#  Report the cluster centers.
#
  print ( '  Kmeans2 cluster centers C:' )
  print ( c )
#
#  Compute the separate energies.
#
  bd = ( xn - c[0,0] )**2 + ( yn - c[0,1] )**2
  rd = ( xn - c[1,0] )**2 + ( yn - c[1,1] )**2
#
#  Sum the blue and red cluster energies.
#
  cost0 = sum ( bd[label==0] )
  cost1 = sum ( rd[label==1] )
  cost = cost0 + cost1
  print ( '  Cluster variance = ', cost, '=', cost0, '+', cost1 )
#
#  Count the blue and red cluster elements.
#
  bn = sum ( label==0 )
  cn = sum ( label== 1)
  print ( '  Cluster size = ',  bn + cn, '=', bn, '+', cn )
```

To check our work, we would like to replot the data, but this time marking the data and cluster centers by color.

```
  plt.plot ( xn[label==0], yn[label==0], 'c.', markersize = 10 )
  plt.plot ( xn[label==1], yn[label==1], 'm.', markersize = 10 )
  plt.plot ( c[0,0], c[0,1], 'bo', markersize = 15 )
  plt.plot ( c[1,0], c[1,1], 'ro', markersize = 15 )
```

and we get a fairly convincing result:



Old Faithful data clustered by `kmeans2()`.