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

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

*Sometimes data reveals more than one central target!*

**Cluster Patterns**

*Machine learning seeks patterns that can simplify or explain data.*

- *totally random data has no explanation;*
- *but often, our data includes many samples of similar structure;*
- *the heights or weights of a class of students might be very close;*
- *we might notice a typical central value;*
- *we might notice a typical variation from this central value;*
- *the normal distribution is a model of this behavior;*
- *if data involves several components, we may need to scale it;*
- *our data might also naturally divide into two or more distinct clusters;*
- *the* **kmeans** *algorithm can partition data into k separate clusters;*
- *the quality of a clustering is measured by its total "energy" E;*

## 1 Sets of 100 data values

Let us consider some artifical sets of $n$ data points, whose typical entry is $(x_i, y_i)$:

- **U**: uniform random values in the unit square;

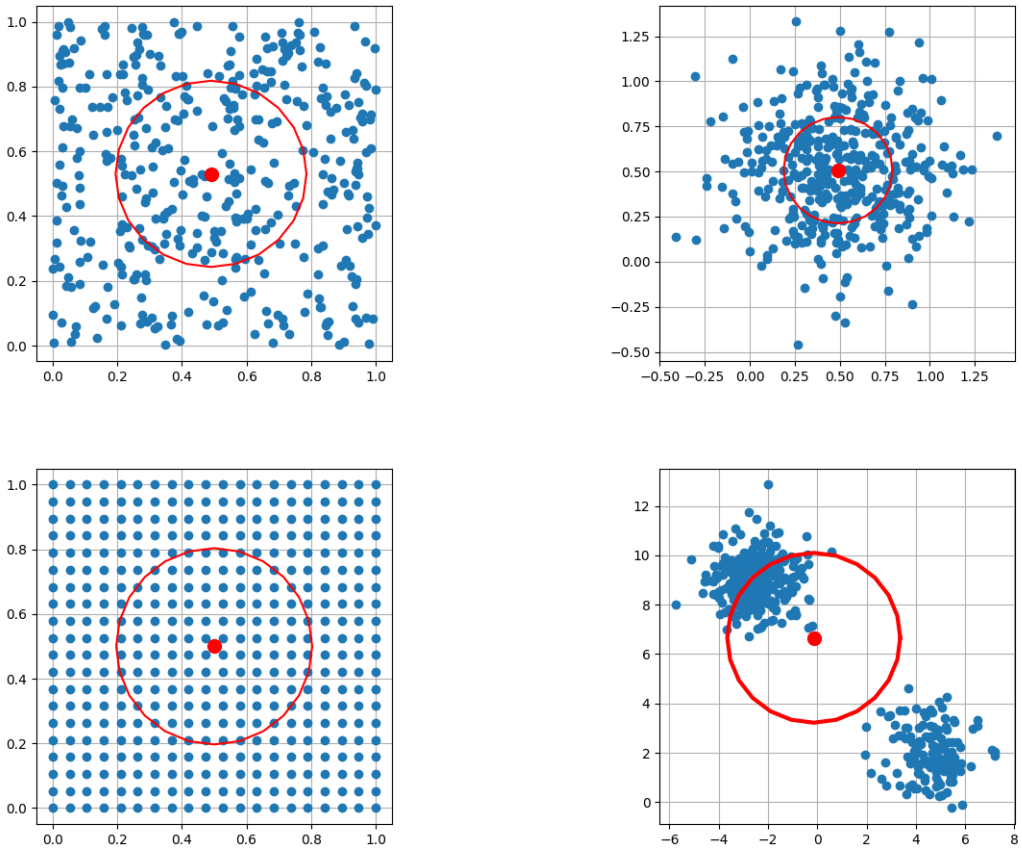
- **N**: normal random values with mean $\mu = 0$ and standard deviation $\sigma = 1$;
- **G**: a $10 \times 10$ grid of points in the unit square;
- **B**: two "blobs", with one blob having twice as many points as the other.

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 $\sigma^2$;
- `np.std(Z)` : the standard deviation $\sigma$.

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.

The purpose of the K-Means algorithm is to investigate such data, guess the number of clusters into which the data falls, sort each data item into the nearest cluster, and then report the total "energy" $E$, which tells us how tightly we have clustered the data.

## 2 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 The Wait Times

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

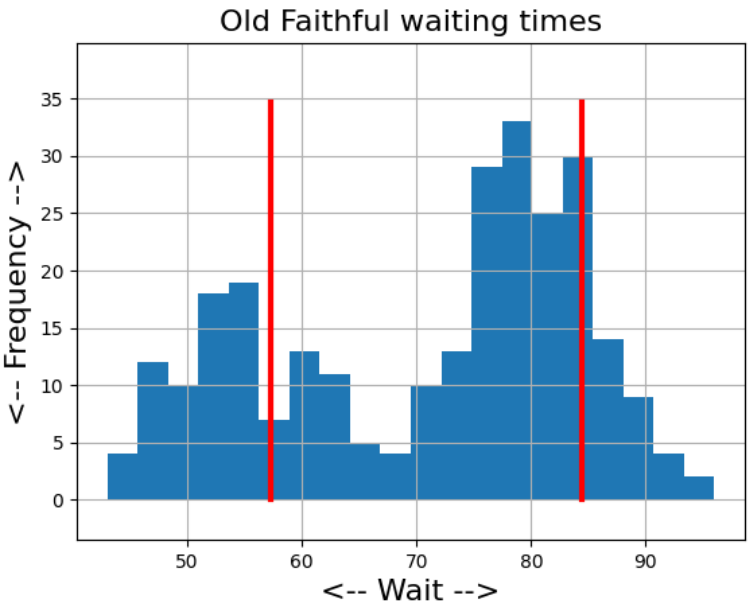
```
y.shape =  (272,)
np.min(y)   =  43.0
np.mean(y)  =  70.8970588235294
np.max(y)   =  96.0
np.var(y)   =  184.14381487889273
np.std(y)   =  13.569960017586371
```

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.

# 4 Considering Duration and Wait Together

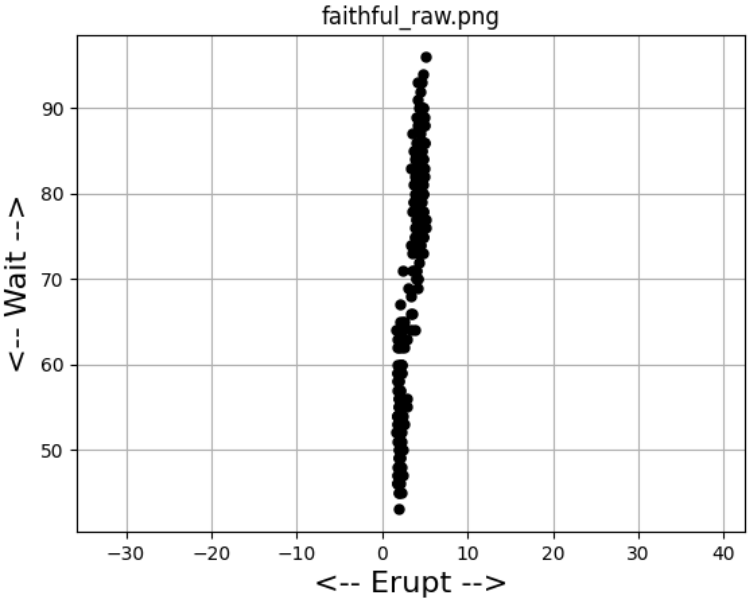
To get a better feel for what is going on, let's consider the bigger picture, that is, the pairs $(x, y)$. First we compute statistics:

```
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.

# 5 Rescaling Data

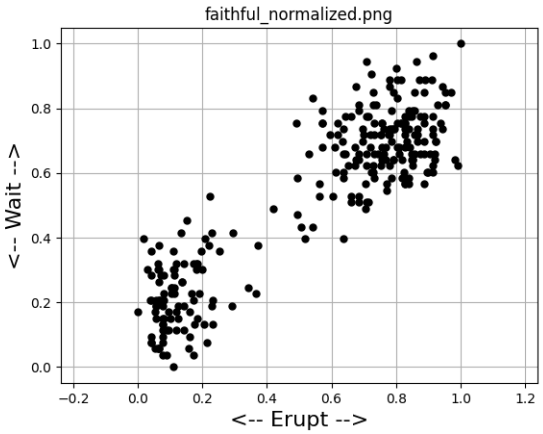
If we *normalize* the data, we will use a linear mapping which preserves the shape of the information, but squeezes or stretches each component separately, to fit into the $[0, 1]$ interval as follows:

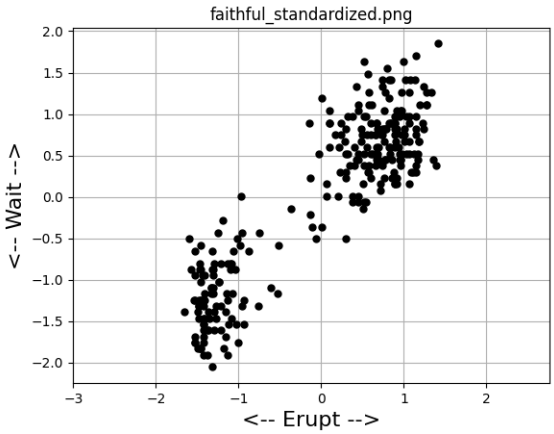
$$xn = \frac{x - \min(x)}{\max(x) - \min(x)} \quad \text{Normalization}$$
$$yn = \frac{y - \min(y)}{\max(y) - \min(y)}$$

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

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

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 )
```

# 6 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[0,0] )**2 + ( data[:,1] - Z[0,1] )**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 )
```

# 7 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 $Z_1$;
- for each data item $i$, computing the distances to either center, $d_0$ and $d_1$;
- 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 the separate energies $e_0$ and $e_1$, and the total E.

For our geyser data, it's easy to estimate two centers Z:

- Center 0: (0.05,0.2)
- Center 1: (0.8,0.75)

It would seem to make sense to assign each data item to the nearest center, that is, for each `i`, we compute

```
d0 = (data[i,0] - Z[0,0])**2 + (data[i,1] - Z[0,1])**2)}}
d1 = (data[i,0] - Z[1,0])**2 + (data[i,1] - Z[1,1])**2)}}
```

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:

$$e_0 = \sum_{C[i]=0} (data[i,0] - Z[0,0])^2 + (data[i,1] - Z[0,1])^2$$

$$e_1 = \sum_{C[i]=1} (data[i,0] - Z[1,0])^2 + (data[i,1] - Z[1,1])^2$$

$$E = e_0 + e_1$$

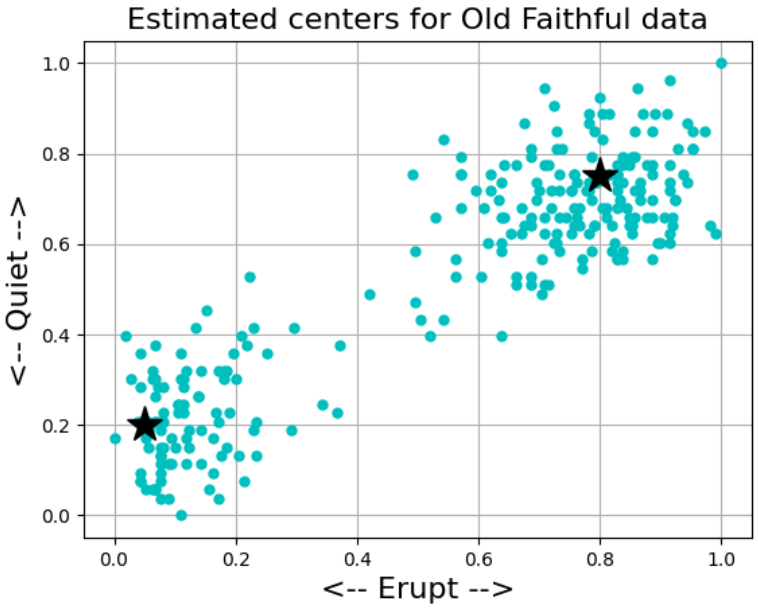
or, in Python,

```
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 $E$ must be less than if 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.

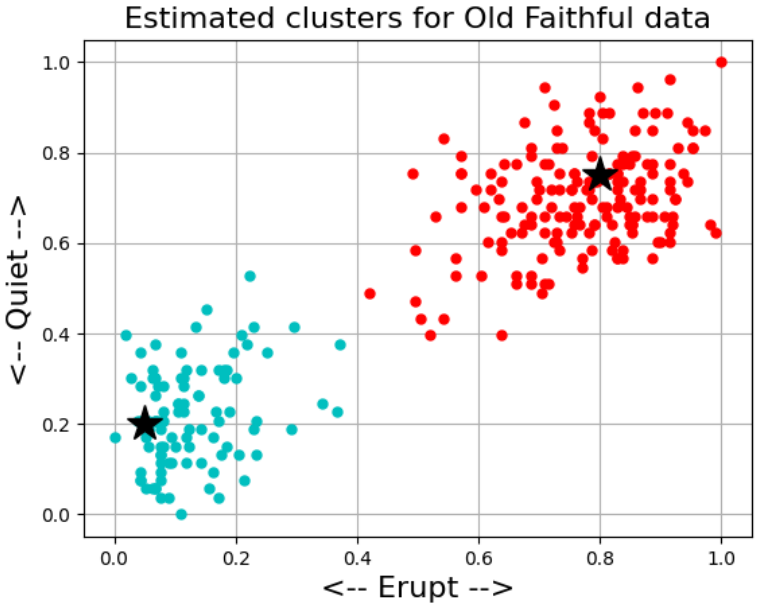
Let's go ahead and see how our guess at cluster centers will work!




*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.

# 8 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 $k$ center, 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,

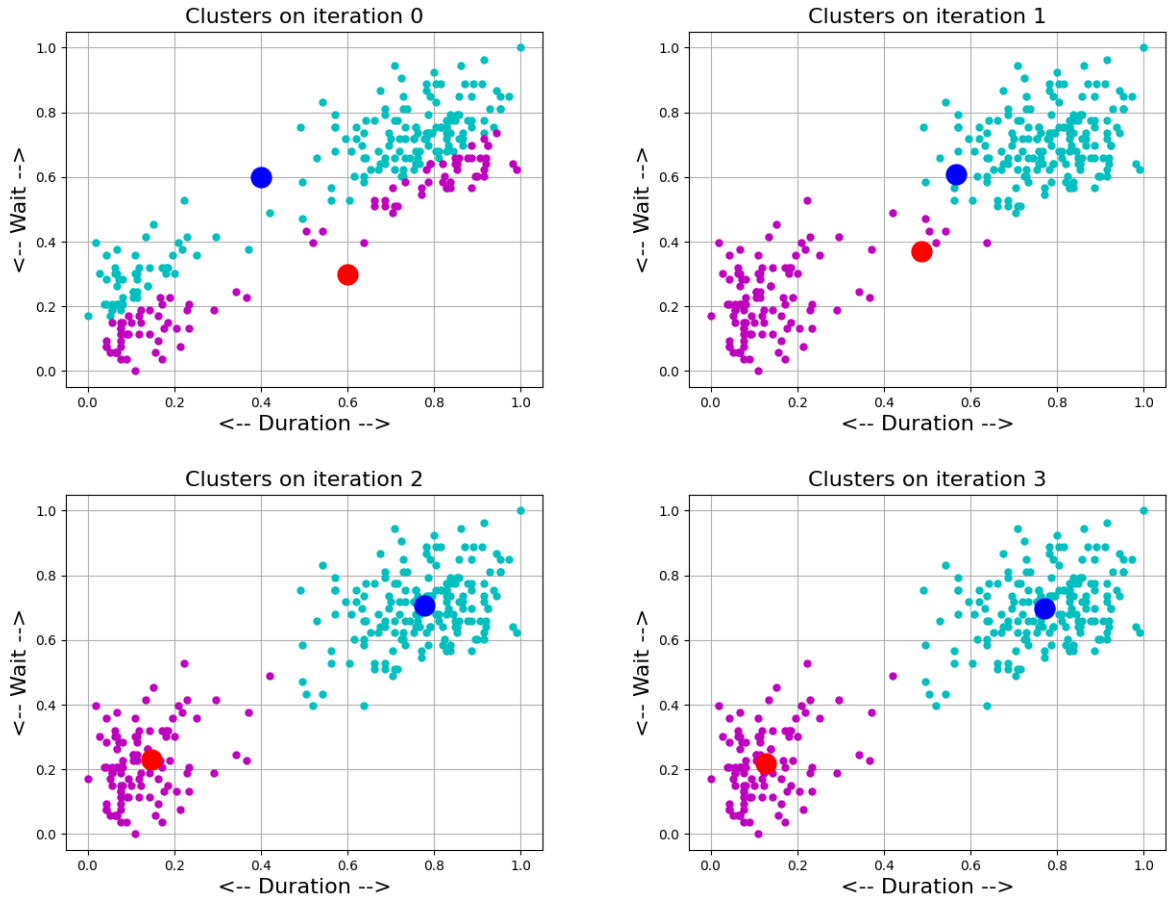
We have already tried an example of these steps by hand, picking some centers and then assigning each data item to the nearest one. Let's think of our example as a suggesting the first step of an iteration. How should we proceed?

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 a version of it is built into the `scipy()` library, and into `scikit-learn`

# 9 Using scikit-learn's KMeans() algorithm

To introduce the `scikit-learn` version of the K-Means algorithm, we will again analyze our Old Faithful geyser data.

To begin with, we need to import the K-Means algorithm from `scikit-learn`:

```
from sklearn.cluster import KMeans
```

Now let's use `np.loadtxt()` to read the file *faithful_normalized.txt*, creating an an $n \times d$ object `data`. Our data is already normalized.

If our data is two-dimensional, now might be a good time to make a scatterplot, so we can take a look and make an guess as to how many clusters `k` there might be, because the choice of $k$ is up to us.

Now, given a choice for the number of clusters, we create a `kmeans()` function to do the job, and we apply it to our data:

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

After the call to `kmeans.fit()`, we have the following information:

- **kmeans.cluster_centers_**: the $k$ centers $Z$;
- **kmeans.labels_**: the cluster assignments $C$;
- **kmeans.inertia_**: the total energy $E$;

Here is a sketch of our Old Faithful program using `scikit-learn`:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.cluster import KMeans

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

k = 2
kmeans = KMeans ( n_clusters = k, n_init = 'auto' )
kmeans.fit ( data )

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

plt.plot ( data[C==0,0], data[C==0,1], 'c.' )
plt.plot ( data[C==1,0], data[C==1,1], 'm.' )
plt.plot ( Z[0,0], Z[0,1], 'bo' )
plt.plot ( Z[1,0], Z[1,1], 'ro' )
plt.show ( )
```

We won't show the plot for this calculation, since it is almost identical to the final plot of the Lloyd iteration shown avove.

This program assumes that $k = 2$. How could we rewrite the program (and even shorten it!) for $k = 3$ or larger values?

# 10 Classifying new data

In machine learning, we classify data by recognizing that it belongs to a particular group. Typically, classification problems begin with a starter set of data items for which the groups are known, such as reference pictures of cats and dogs.

When we are using clustering, we start out with no such information, just the raw data. However, once we have carried out the K-Means algorithm, we have actually created a model of the data that can be used to classify new items. That is, given a new piece of data, we can simply assign it to the group whose center is nearest to the data.

The `scikit-learn` function `kmeans()` includes a `predict()` function which allows us to do this very easily. Assuming you have already set up your clusters, let's suppose you have some new samples which we will store as `data2`. To classify this new data, that is, to determine the appropriate cluster assignments, simply issue the command

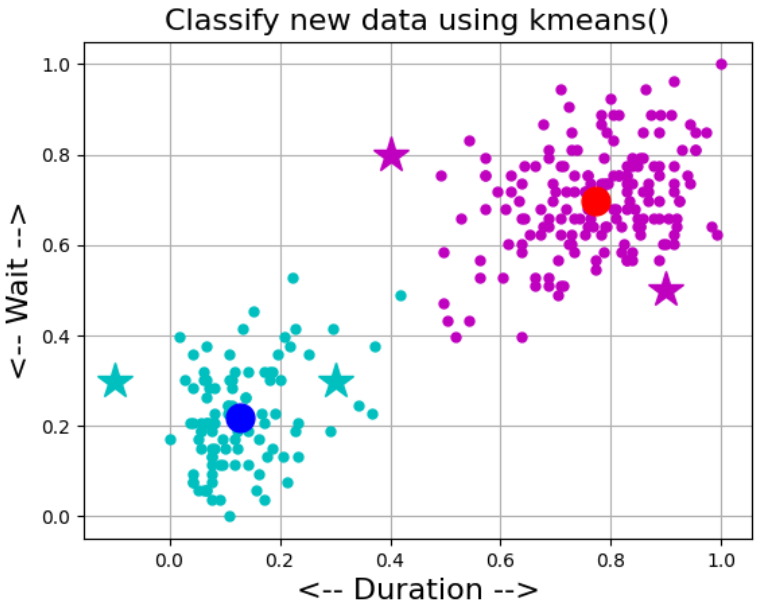
```
\item{ {\tt{C2=kmeans.predict(data2)}};}
```

The new cluster assignments will be stored C2, and can be used as follows:

```
data2 = np.array ( [
  [-0.1, 0.3 ],
  [ 0.3, 0.3 ],
  [ 0.4, 0.8 ],
  [ 0.9, 0.5 ] ] )

C2 = kmeans.predict ( data2 )

plt.plot ( data[C==0,0], data[label==0,1], 'c.' )
plt.plot ( data[C==1,0], data[label==1,1], 'm.' )
plt.plot ( data2[C2==0,0], data2[C2==0,1], 'c*' )
plt.plot ( data2[C2==1,0], data2[C2==1,1], 'm*' )
plt.plot ( Z[0,0], Z[0,1], 'bo' )
plt.plot ( Z[1,0], Z[1,1], 'ro' )
```




The stars indicate the locations of the new data items we just classified.

## 11 Choosing the value of $k$

The K-Means algorithm requires you to specify a value for $k$, the number of clusters. For the geyser problem, we were able to assume that $k = 2$ was a good choice, simply by plotting our rather small set of data. But what happens when there is a lot more data, or it is 5, 10, or 20-dimensional?

The key to finding a reasonable value of $k$ is to concentrate on the behavior of the energy $E$. In the geyser problem, we saw that going from $k = 1$ to $k = 2$ resulted in a significant drop in $E$. Note that $E$ must always be nonnegative, and that it should decrease as $k$ increases, until $E$ hits the value of zero when $k = n$ (because every data item will be its own center).

If we didn't know that the geyser data formed two clusters, would the behavior of $E$ warn us, as we increased $k$? To answer this question, suppose we simply compute $E$ as $k$ runs from 1 to 10. Here's how we would do that:

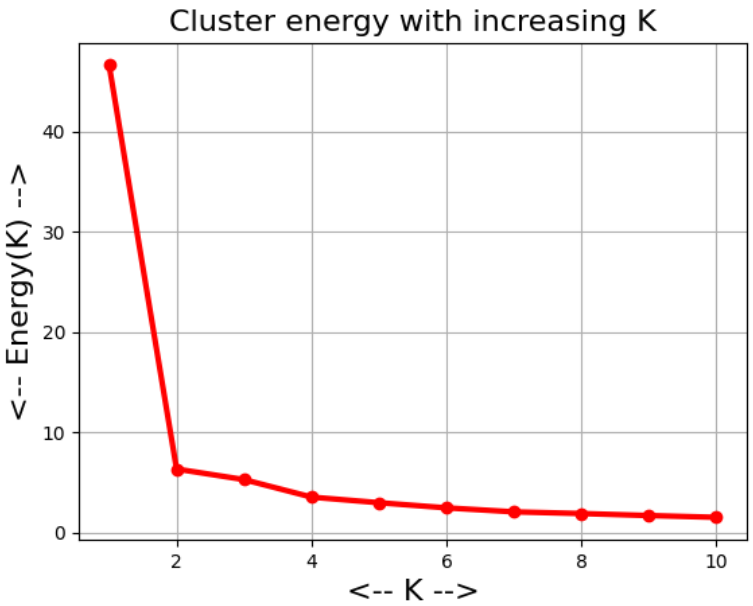
```
data = np.loadtxt ( 'faithful_normalized.txt' )

kmax = 10
kplot = list ( range ( 1, kmax + 1 ) )
eplot = np.zeros ( kmax )

for i in range ( 0, kmax ):
  k = kplot[i]
  kmeans = KMeans ( n_clusters = k, n_init = 'auto' )
  kmeans.fit ( data )
  eplot[i] = kmeans.inertia_

plt.plot ( kplot, eplot, 'r-o', linewidth = 3 )\\
```

and here is the resulting plot:




We can clearly see the huge drop going from $k = 1$ to $k = 2$, after which increasing $k$ doesn't seem to produce further signficant drops in $E$. So for this data, a good guess would be that two clusters is enough.

In the lab exercises, we will perform a similar test for which the data seems to fall into a larger number of clusters.

## 12 Using the Model for Simulation

Our K-Means algorithm has allowed us to create a 2-cluster model of the geyser data, with given populations, centers and energies. If we wish, we can use this model in a simulation, which will return as many values as we like, based on the statistics we have estimated from the data.

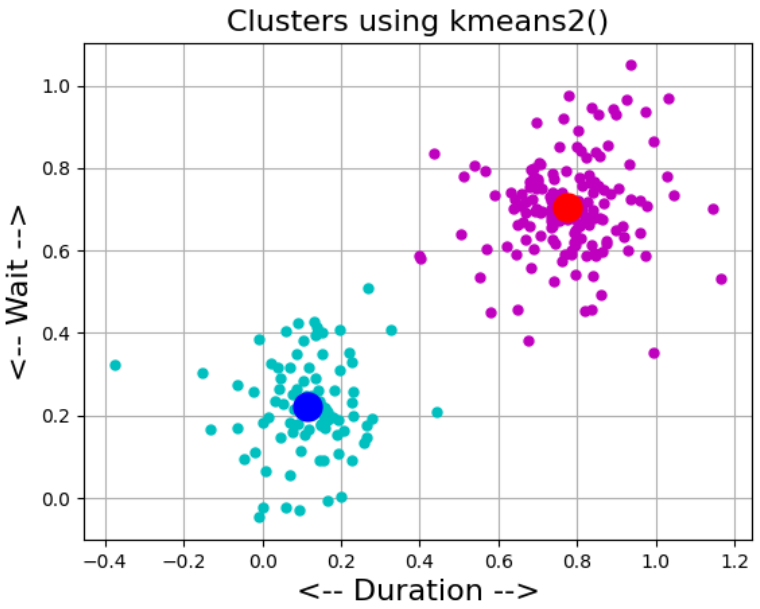
The simulation process goes something like this. We observed $n = 272$ items, with $n_0 = 98$ in cluster 0, and $n_1 = 174$ in cluster 1. Therefore, to simulate one new item, we will use a random uniform value to choose the cluster, with weights of $\frac{98}{272}$ and $\frac{174}{272}$ respectively.

Suppose we have chosen cluster 0. We estimated this cluster to have center at $Z_0 = [0.128, 0.219]$, and an energy of $e_0 = 1.883$. That means the standard deviation is $sigma = \sqrt{\frac{e_0}{n_0}} = 0.1386$.

To choose a random point with this center and standard deviation, we now pick a random normal number $u$, a random angle $\theta$, and set

```
x = Z[0,0] + sigma * u * np.cos ( theta )
y = Z[0,1] + sigma * u * np.sin ( theta )
```

If we use this idea to compute a new set of 272 pretend geyser data values, the resulting plot is:




The plot is similar to our true data, which may be good enough for our purposes. However, it is a limited approximation. For two things, our generated data tend to be symmetric around the two centers, and it could occasionally include negative values, which cannot be correct.