Cluster Patterns in Data
ML_2022: Machine Learning

https://people.sc.fsu.edu/~jburkardt/classes/ml_2022/cluster_lecture/cluster_lecture.pdf

Sometimes Nature takes shots at a target, but her aim isn't perfect!

Cluster Patterns

Given new data, we can search for is evidence of a central tendency, or clustering.

- *clustering is a simple pattern common to many phenomena;*
- *there may be a "target" or central value;*
- *individual examples "aim" for the target, but tend to miss by a typical amount;*
- *the normal distribution is a model of this behavior;*
- *in more complicated examples, we may need to adjust for scale factors;*
- *we may also discover that there are several targets, associated with separate but related processes;*
- *the* **kmeans** *algorithm can partition data into* `k` *separate clusters;*

1 A Model of Clustering in One Dimension

We are used to the idea that there is an average height, an average number of children per family, an average income, an average number of miles traveled by car each year. The average number of children, as computed in 2019, is 1.93, and we would NOT expect any family to have exactly that number. We simply regard the average as some kind of "target" or "central value", so that actual values usually tend to be close (and, in this case, integers!).
In some cases, such as for the height of men 20 years and older, we can say more. Measured values of height will tend to be near the average value of 5 feet and 9 inches (1.75 meters) in a way that approximates a mathematical shape called the **normal distribution**. The normal distribution, with parameters μ and σ^2, determines the probability of a value x as:

$$N(x;\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

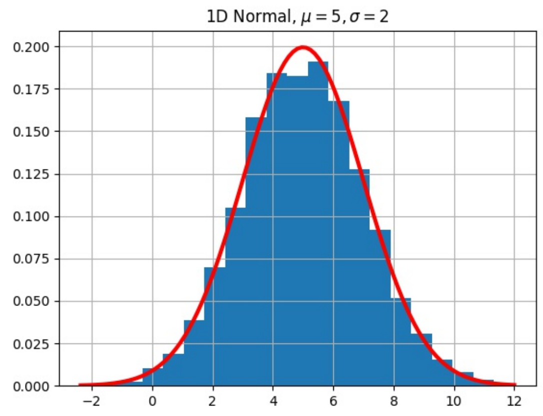
Here μ is the **mean** or **average value** and σ^2 is the **variance**. The square root σ is known as the **standard deviation**. A plot of the normal distribution will show why it is known as the "bell curve", since it has a bell-shaped central peak. Changing the value of μ simply moves the curve left or right, so that its peak value is at $x = \mu$. Changing the value of σ^2 makes the curve wider or narrower, and indicates how strongly the values x tend to cluster near the average.

We can create simple computational models of a process that obeys the normal distribution by using the `numpy()` function `np.random.rand(loc=?,scale=?,size=?)` where

- **loc** is the mean value, μ, defaulting to 0;
- **scale** is the standard deviation, σ, defaulting to 1;
- **size** is shape of the result, defaulting to 1;

We can generate 1,000 normal random values, and then create a histogram of the result:

```
  x = np.random.normal ( loc = 5, scale = 2, size = 10000 )
  plt.hist ( x, bins = 21 )
```



A histogram of 10000 normal random values

2 Clustering in Higher Dimensions

In some cases, the quantities being measured have two or more dimensions. We may still find that these measurements have a clustering behavior similar to what we saw for some one dimensional data. However, there is an interesting complication. For a d-dimensional problem, the normal distribution parameters now become:

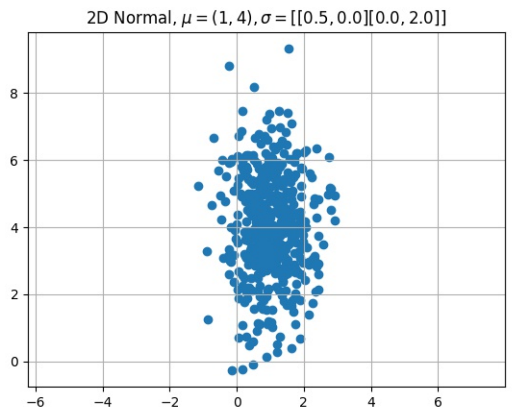
- μ, the average value of each component, a d vector;
- Σ^2, the covariance, a $d \times d$ matrix;

The diagonal entries $\Sigma^2_{i,i}$ are similar to the variance we saw in the one-dimensional case, measuring a tendency to spread or cluster along the i-th variable. An off-diagonal entry $\Sigma^2_{i,j}$ measures the correlation between variables i and j. If is zero, the two variables do not influence each other. If it is positive, the two variables have a tendency to be large or small together; a negative value indicates that a large value of one variable tends to be associated with a small value of the other. We will see some examples to explain this idea.

Another thing to note is that the matrix Σ^2 must be symmetric and positive definite. Symmetric simply means that, for every pair of indices i and j, it must be the case that $\Sigma^2_{i,j} = \Sigma^2_{j,i}$. Being positive definite is a technical condition. It will probably not help you right now to say that it is equivalent to requiring that all the eigenvalues of Σ^2 must be positive.

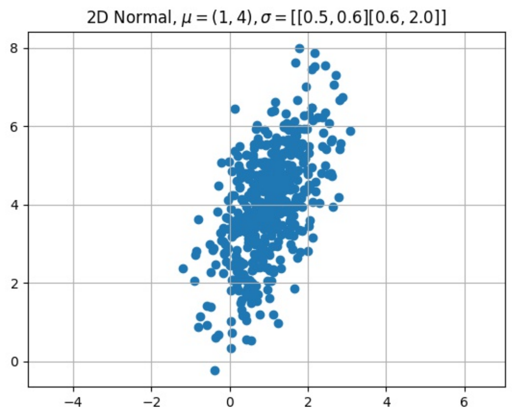
If we work in two dimensions, then we can use the `numpy()` function `np.random.multivariate_normal()` to generate sample values. For simplicity, we will start with a covariance matrix that is diagonal. In this case, we say the variables are **independent**, and do not influence each other.

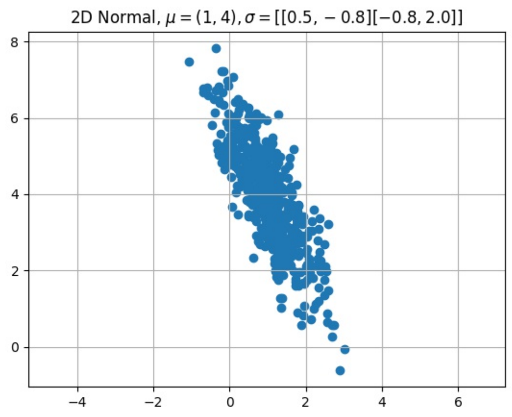
```
mu = np.array ( [ 1.0, 4.0 ] )
cov = np.array (  [ [ 0.5, 0.0 ], \
                    [ 0.0, 2.0 ] ] )
n = 500

xy = np.random.multivariate_normal ( mu, cov, n )

plt.scatter ( xy[:,0], xy[:,1] )
```



A histogram of 500 (independent) normal random values

Now let's get a little idea of what the off diagonal elements of Σ^2 do. We start with our original diagonal matrix andn put a positive or negative value in the off-diagonal position. We see that the positive value tilts the plot so that x and y tend to grow or decrease together, while the negative value produces the opposite effect.





Histograms of 500 positive and negatively correlated normal random values

These experiments suggest that the normal distribution can be used to model the behavior of data that tends to cluster around a mean value. For one dimensional data, the variance describes the tightness of the clustering. In higher dimensions, the covariance matrix is necessary, whose diagonal elements describe the tightness of the clustering, while the off-diagonal elements give the correlation between pairs of dimensions of data.

In lecture #2, we discussed how to compute the mean and variance of one-dimensional data, either by calling the `numpy()` functions `mean()` and `var()`, or by direct calculation of formulas. For the multidimensional case, here are the definitions and the helpful functions. We assume that `x` is now an $n \times d$ array, where each of the n rows contains a d-dimensional data item.

$$\mu_i = \frac{1}{n} \sum_{r=0}^{r<n} x_{r,i}$$

$$\Sigma^2_{i,j} = \frac{1}{n} \sum_{r=0}^{r<n} (x_{r,i} - \mu_i)(x_{r,j} - \mu_j)$$

although, again for technical reasons, the fraction multiplying the sum in the calculation of Σ^2 is usually chosen to be $\frac{1}{n-1}$.

In `numpy()` we can easily compute both quantities:

```
mu = np.mean ( x, axis = 0 )         # You MUST include the axis argument!
covar = np.cov ( x )                 # Uses 1/(n-1) multiplier, by default
covar = np.cov ( x, bias = True )    # uses 1/n multiplier
```

We will use some of these concepts in the next lab.

3 When Nature Hands Us Two Clusters

Clustering is a simple kind of pattern that occurs often. However, in many cases, data will have a choice of several centers around which to cluster; a plot of two-dimensional data with such behavior will show several blobs, which may be far apart, or somewhat overlapping. If we think such a pattern is meaningful, then we probably want to describe it, and be able to state that there are a given number k of clusters, each with a specific center value. We will want to be able to state which cluster the i-th data item belongs to. And we may even want to able to predict, given a new data item, which cluster it should go to.

As an example of this situation, we will look at the Old Faithful geyser. Folklore suggests that this geyser erupts "faithfully" every hour, but the actual behavior is more complicated. There is a somewhat irregular schedule, in which the geyser erupts for `E` minutes, and then remains waits for `Q` minutes. Here, `E` is much smaller than `Q`, and `Q`, by the way, is not roughly 60 minutes!

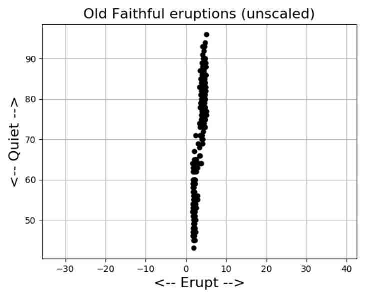
The data file *faithful_data.txt* contains 272 pairs of observations `(E,Q)`. By making a scatterplot of the data, we can look patterns. Remember that, even when the `x` and `y` data are of greatly different scaleds, plotting usually hides this discrepancy by automatically scaling.

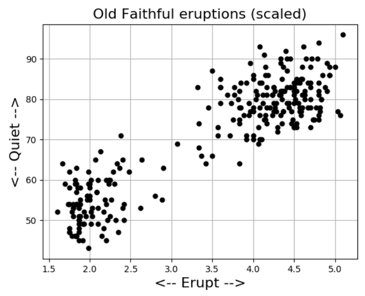
```
plt.plot ( x, y, 'k.' )
plt.xlabel ( '<-- Erupt -->' )
plt.ylabel ( '<-- Quiet -->' )
plt.title ( 'Old Faithful eruptions (unscaled)' )
plt.grid ( True )
plt.axis ( 'equal' )  # Use the same scale on x and y axes
plt.show ( )
```

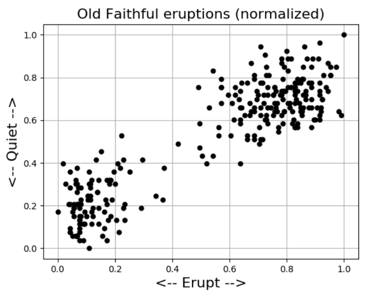
Listing 1: Scatterplot of Old Faithful observations

Notice the command `plt.axis ( 'equal' )`. This forces the plotting program to use the same scale on both axes, and results in the horrible "unscaled" plot. Removing that line gives us a much more interesting "scaled" plot. However, we have been warned. This data has a huge difference in the `x` and `y` scales, and we need to watch out for this. The right approach is to normalize the data:

```
x = ( x - np.min ( x ) ) / ( np_max(x) - np.min(x) )
y = ( y - np.min ( y ) ) / ( np_max(y) - np.min(y) )
```







Old Faithful Data, unscaled, scaled, and normalized versions

4 A Guess at Clustering

Now the purpose of our investigation is to try to describe any patterns we see in the data. If we simply compute the mean and variance, we do get a description of the data, in terms of clustering around a single center. But that's really not what we see. Instead, if asked to describe the plot, we'd say some points cluster around one center, and some around the other. That suggests that we could describe this pattern by

- identifying two separate centers or "means";
- assigning each data item to a particular center, or "cluster";
- estimating the typical distance of a data item to its center, or "cluster variance"

Working with our normalized data, it's easy to estimate the location of two centers:

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

to be stored in an array `C`.

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

- `d0 = (x[i,0] - C[0,0])**2 + (x[i,1] - C[0,1])**2);`
- `d1 = (x[i,0] - C[1,0])**2 + (x[i,1] - C[1,1])**2);`

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

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

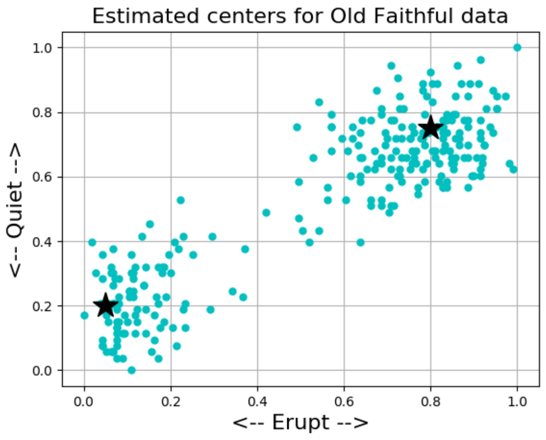
And we can compute the variance of each cluster:

$$\text{var}_0 = \frac{1}{n_0} \sum_{y[i]=0} (x[i,0] - C[0,0])^2 + (x[i,1] - C[0,1])^2$$

$$\text{var}_1 = \frac{1}{n_1} \sum_{y[i]=1} (x[i,0] - C[1,0])^2 + (x[i,1] - C[1,1])^2$$

Notice that, unless we did a terrible job picking the two centers, $var_0 + var_1$ must be less than the original variance of the data, before we did any clustering. Essentially, this is because the variance is like requiring everyone in the US to travel to Washington, DC, while the cluster variance is asking each person to drive to their state capital. In the second case, the total distance driven is much much less!

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



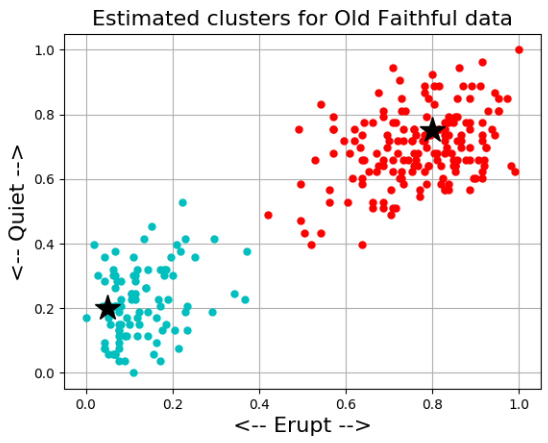
Old Faithful Data, with rough guess at two cluster centers

Once we have chosen our centers, we simply need to cluster each data item with the nearest center. Here's one way to do that:

```
#
#  Compute distance to each center
#
  d = np.zeros ( [ n, 2 ] )
  d[:,0] = ( C[0,0] - data[:,0] )**2 + ( C[0,1] - data[:,1] )**2
  d[:,1] = ( C[1,0] - data[:,0] )**2 + ( C[1,1] - data[:,1] )**2
#
#  Y assigns data to nearest center
#
  y0 = np.where ( d[:,0] <= d[:,1] )
  y1 = np.where ( d[:,1] < d[:,0] )
```

Now, `y0` and `y1` are lists of the data belonging to clusters 0 and 1 respectively. We can display the clustering using commands like

```
  plt.plot ( data[y0,0], data[y0,1], 'c.', markersize = 10 )
  plt.plot ( data[y1,0], data[y1,1], 'r.', markersize = 10 )\\
```



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

Finally, we can compare the one cluster variance to the two cluster variance. We get the following results

```
  1 Cluster size  272              1 Cluster variance = 0.172
  2 Cluster size  272 = 97 + 175   2 Cluster variance = 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 variance.

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. What happens with bigger data sets, higher dimensions, and possibly more clusters?

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

5 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 must produce

- a set of k centers C;
- an assignment of each x_i to some cluster C_j so that $y[i] = j$;

We have already done an example of these two steps by hand, picking some centers and then assigning each x to the nearest one. Let's think of out example as a first step. So is there any thing more we can do?

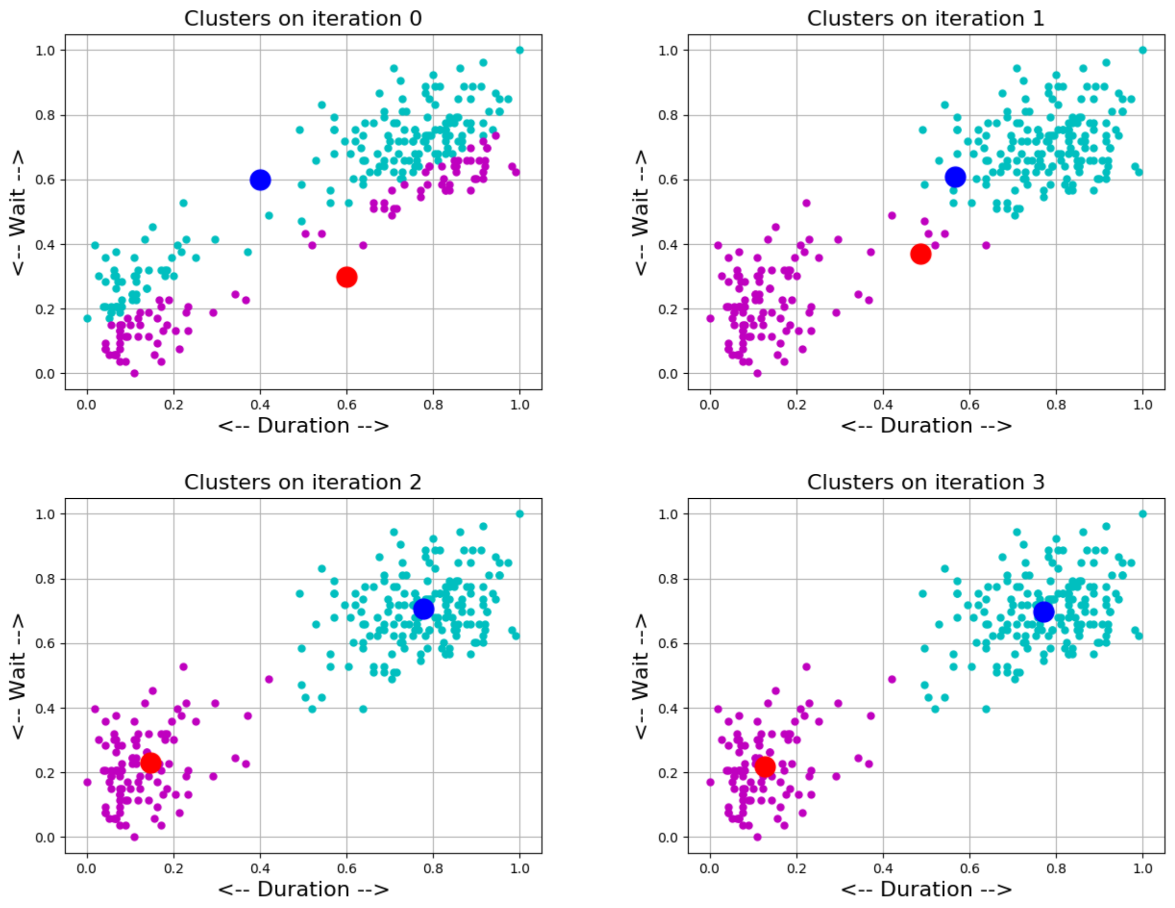
1. The centers should be the means of the data that is assigned to them. But the centers were chosen first, then assignments were made. Update the centers by replacing them by data averages;
2. The data assignments were made to the old centers. Now the centers have changed. Recompute data assignments to nearest centers;

Now you might think that second two-step sweep would be enough. But actually, each time we do one step, it turns out we need to follow it with the other step, so that we are repeatedly updating the centers and assignments. This means that the K-Means algorithm is an *iteration.*

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

6 Using scikit-learn's KMeans() algorithm

As you might have guessed from our initial exercise, it's possible to program a reasonable version of the K-Means algorithm yourself. However, we will rely on `scikit-learn`, and discuss how we would apply K-Means to our data about the Old Faithful geyser.

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

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

Now let's assume that we have used `np.loadtxt()` or `pd.read_csv()` to read `x`, an $n \times d$ set of numeric data items. Since our data may be badly scaled, we now apply `normalization` or `standardization` to make the components of the data more comparable.

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.

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 )
y = kmeans.fit_predict ( x )
```

Here, the output `y` is a vector of length `n`, such that, for each index $0 \leq i < n$, the data item `x[i]` has been assigned to cluster `y[i]`.

If we have two-dimensional data, and we have three clusters, we could plot them by:

```
plt.scatter ( data2[y==0,0], data2[y==0,1], c = 'red' )
plt.scatter ( data2[y==1,0], data2[y==1,1], c = 'cyan' )
plt.scatter ( data2[y==2,0], data2[y==2,1], c = 'green' )
```

Along with the cluster assignments, some other information is created which may be of use. Note that these variables start with the word `kmeans` or whatever name you gave to your algorithm when you called `KMeans()`. Also, they have a final underscore in the name, because they are not "official" output variables, visible in the function call.

- `kmeans.cluster_centers_` is a $k \times d$ array of cluster centers;
- `kmeans.inertia_` is the total cluster variance;

Now suppose you have some new samples `x2` to be added to your dataset. You could, of course, simply combine this data with your old data and recluster everything. That would mean that the cluster centers would probably change, and even some of the old data might switch its assignments. Or, you may decide to leave the original clustering fixed, and simply assign the new data to whichever center is closest. If that's what you want to do, then you use the following command to get the assignments of the new data:

- `y2=kmeans.predict(x2)`;

7 Cluster variance, and the number of clusters

We have already seen that, in statistics, the covariance matrix `cov=np.cov(x)` of a set of d-dimensional data x gives us a $d \times d$ array of information, with the diagonal entries being variances. For clustering problems, we want to keep track of the *clustering variance*, which is the sum of the d diagonal variances. In linear algebra, the sum of the matrix diagonal entries is known as the `trace`, and `numpy()` provides a function to compute this:

```
cov = np.cov ( x )
v = trace ( cov )
```

This value `v` represents the inertia of the initial set of data, before we do any clustering. Essentially, it takes the mean of the data as the cluster center `C`, and then sums the squares of the distance of each data item. We can also get this quantity by applyhing a `kmeans()` procedure to out data with `k=1`, and then printing the value of `kmeans.intertia_`.

It should be obvious that, if we now do a two-clustering of the same data, and compute the intertia, we must get a lower value; if we go to three clusters, the value will again decrease, and it will keep on decreasing until we get to the (ridiculous) value of `k=n`.

As it turns out, if we want help in choosing an appropriate value of `k`, perhaps because our data is of higher dimension, then a good idea is simply to try a sequence of values of `k`, and print or plot the inertias. Often, the plot will decrease steeply for a while, and at some value of `k`, it will level off, something like a hockey-stick. This value of `k` can be assumed to be a natural choice for the number of clusters.