Clustering Analysis

Application in Python

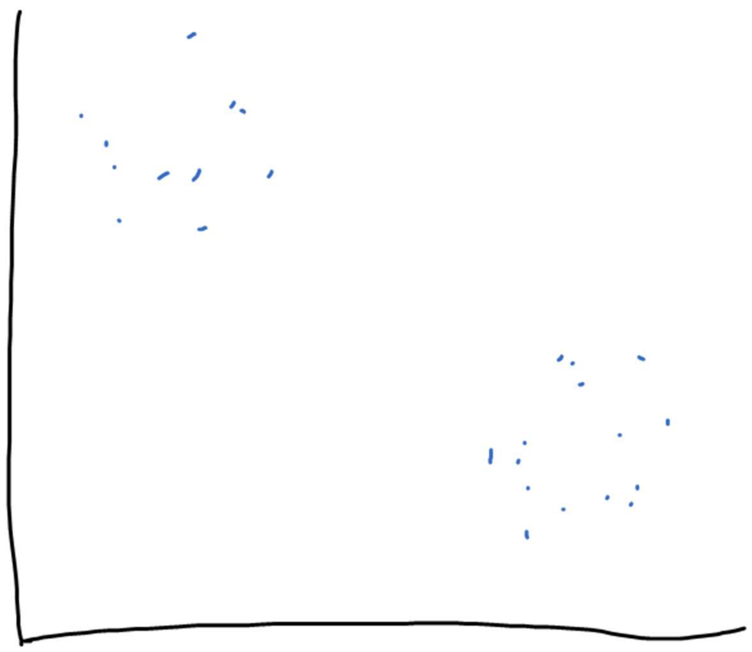
James Conlon
CME 594 Data Science
University of Illinois at Chicago

Introduction

It is assumed that the reader has a basic understanding of the mathematics behind clustering, but we will start with a brief overview of clustering

Clustering is a type of unsupervised machine learning. The end goal is to group (*cluster*) data points together in a way that fits the data set as a whole. Unlike classification or supervised machine learning, the given data set in clustering does not have classification (*labels*).

How do we decide which data points to cluster?

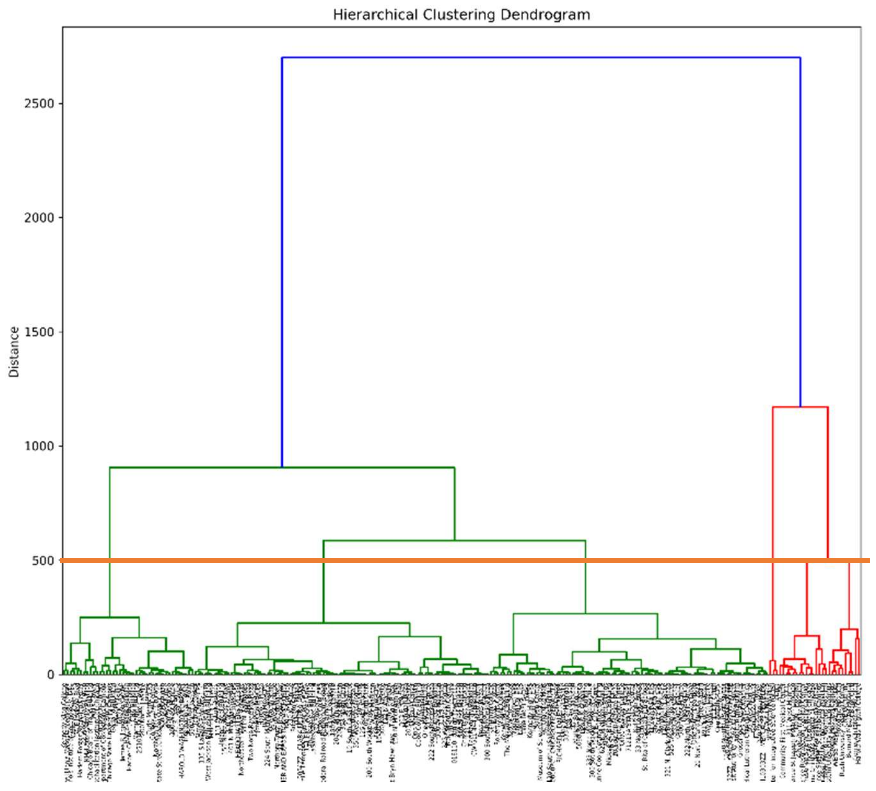


Let's name this data set *X*. Notice how the axis on this plot are not labeled. *X* contains both the 'x' and 'y' coordinates and can be thought of as a two-column data table. One column contains values plotted on one axis, and the second column contains values on the other axis. Thinking of data in this way will be beneficial further in this reading when applying this technique in Python.

In this example plot, it is easy to recognize two separate clusters. When looking at real data, it is unlikely to find something as clear as this. It is not always obvious which points belong to which cluster. It helps to understand cluster proximity and distance when deciding which cluster to assign in hierarchical clustering

Recall the dendrogram that can be created when a hierarchical clustering algorithm is used. It shows the order that individual points are clustered and their resulting distances. To find n-clusters, we can specify in the algorithm when to stop pairing points. Another way to find clusters in hierarchical clustering is using an algorithm that limits distance. This is an example of density-based clustering and will be covered further in this reading.

The line on the dendrogram below illustrates how a density or proximity based clustering algorithm would work on a limitation of 500 distance units

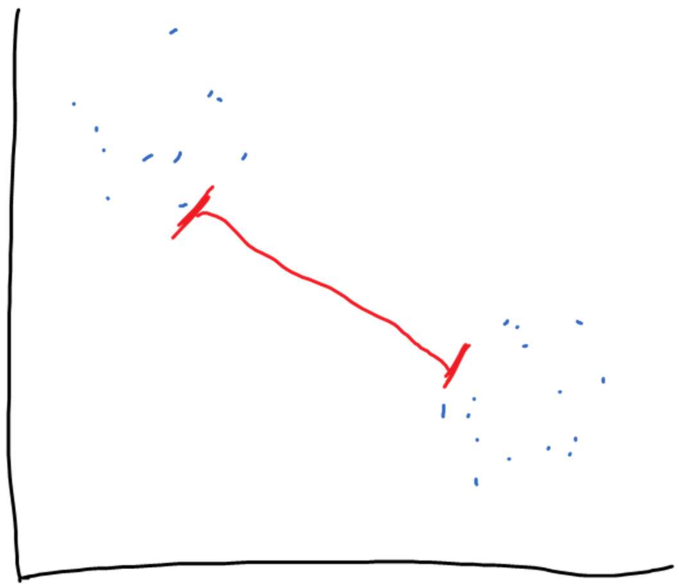


How is distance measured in clustering? It depends on the Linkage type. The main types of linkage are

- Single
- Complete
- Average
- Mean

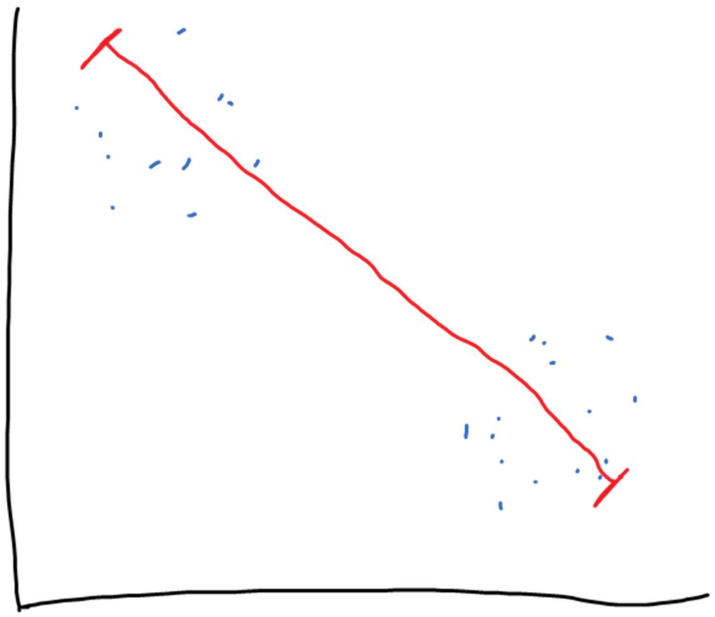
Linkage Types

Single Linkage



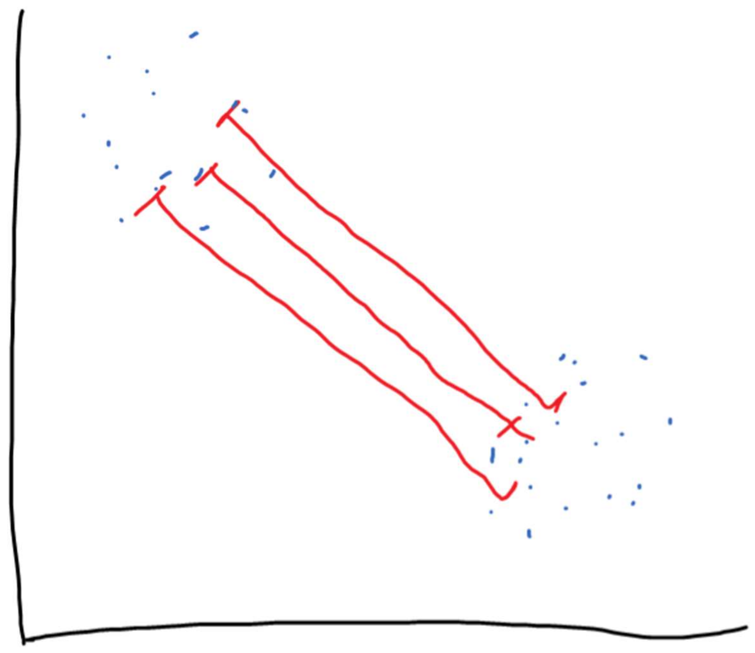
The single linkage distance is the Euclidian distance of the two closest points between clusters. This is similar to the measurements used in the K-Nearest Neighbor algorithm. Single linkage can be used to minimize cluster span.

Complete Linkage



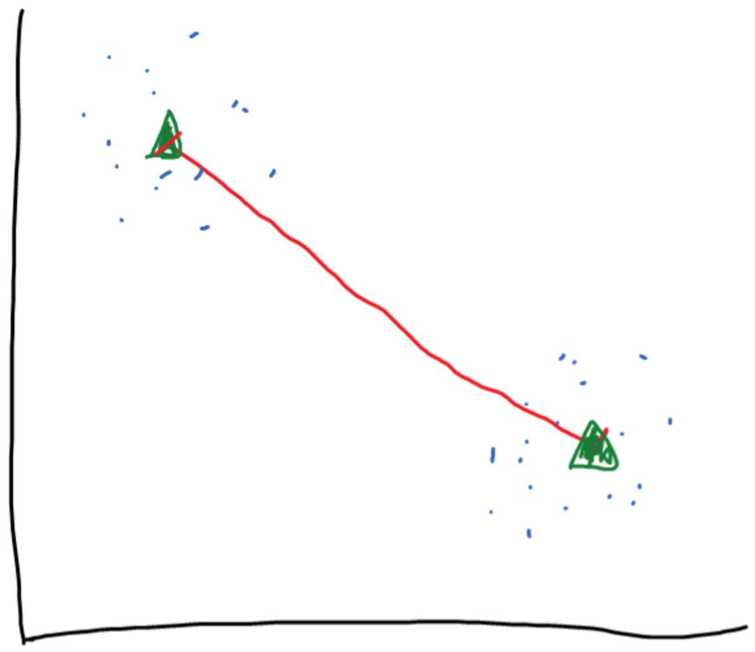
Complete linkage is the distance of the two farthest points between clusters

Average Linkage (Average of distances)



Average linkage is the result of taking the all distance measurements of two clusters and averaging them. This is not to be confused with **mean** linkage.

Mean Linkage (Distance of Averages)



Mean linkage is the distance between the centroids (green triangles) of two cluster

k-Means

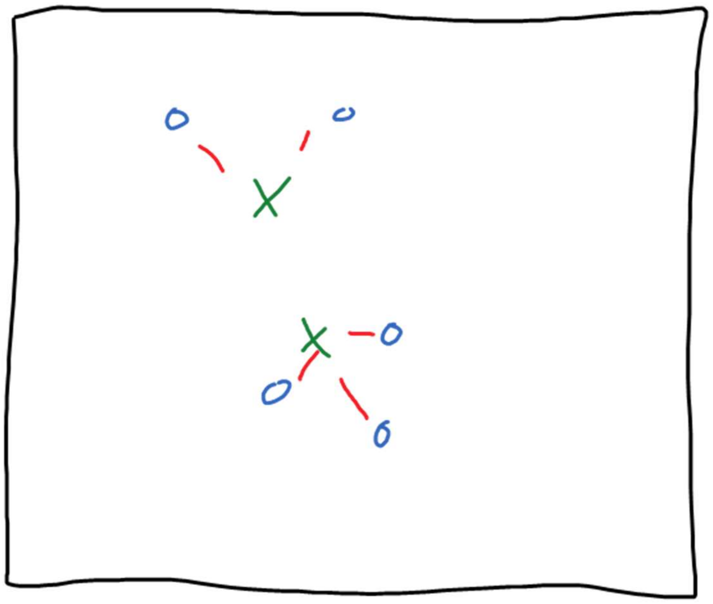
The k-means algorithm is easily applied in Python (through scikit-learn). The basics behind k-means are easy to understand.

Let's look at a simple k-Means examples with $k = 2$

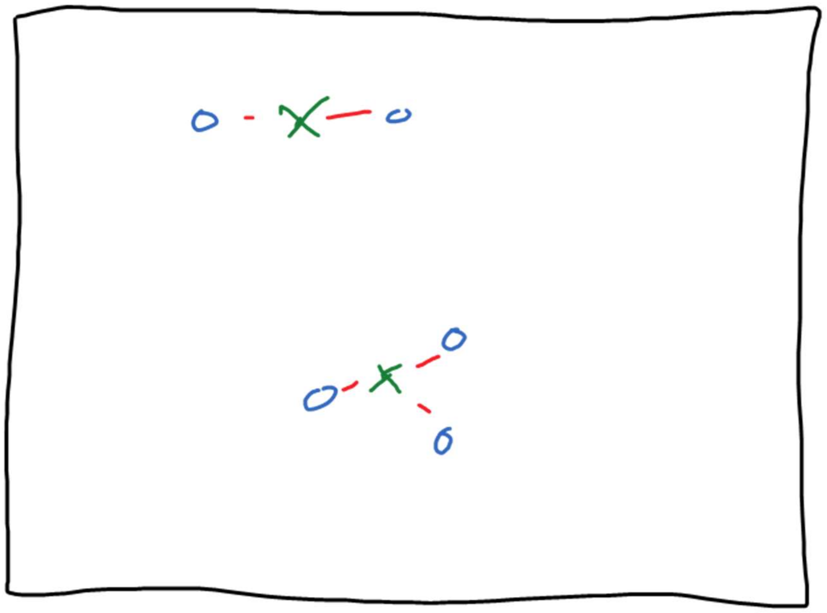
There are two main steps in the k-Means algorithm:

1. k starting points are initialized. The starting points can be either randomized based on the data's spread or calculated using *K-means++* (which is more complicated but designed to improve clustering performance). For sake of the example, let's assume that the initialization is randomized.

   In figure (1), the blue circles are data points and the two green crosses are the randomly assigned initial cluster locations. Each of the points are assigned to the closest cluster.



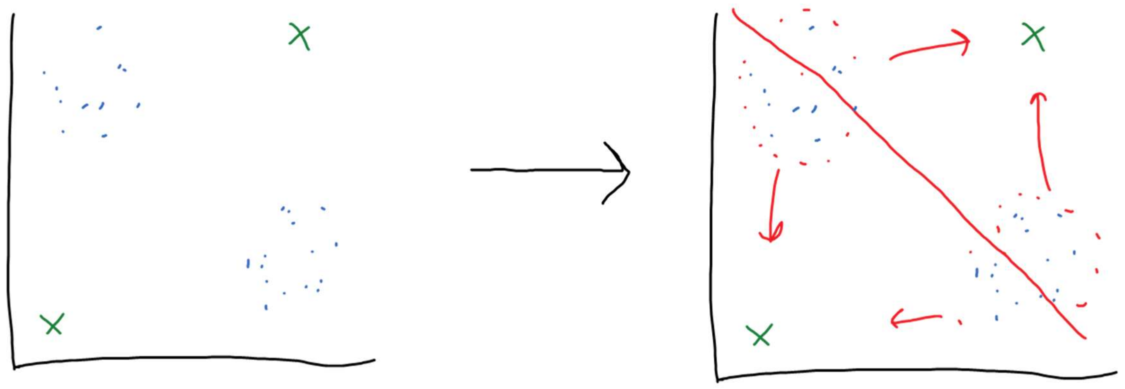
2. The centroid of each cluster is calculated and the cluster location is moved to the centroid



The distances between points and clusters are recalculated. Points are assigned to a cluster. In this case, no point changes clusters. The algorithm has finished.

If any point becomes closer to a different centroid, step 1 is repeated. This is continued until each point is assigned to its closest cluster.

It is important to note that this algorithm is guaranteed to converge. This means that it will always return *k*-clusters with each point assigned to a cluster. It is *not* guaranteed to give the optimum fit for the data given. The following is an example of how it is possible for k-Means to return a less than desirable result.



Based off the initialization, the two clusters converge, but result in a bad representation of the data…

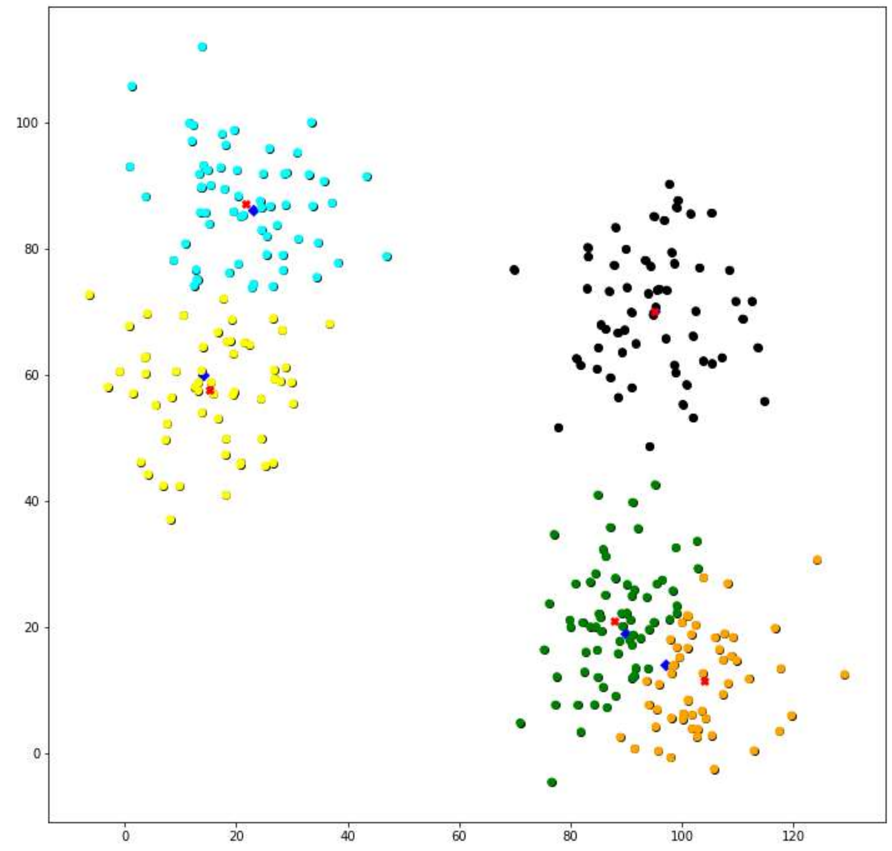
Implementing in Python

All Python scripts and images used in this reading are available for download on GitHub and can be found by following this link: https://github.com/jamesconlon/k-means-clustering

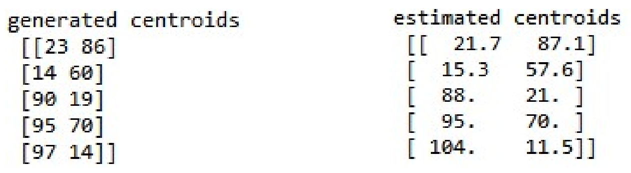
The first example is a demonstration of how k-Means can be used to cluster data. Random clusters are generated through scikit-learn's samples generator. K-Means is then used to find clusters in this random data. The script can be tweaked to change the number of clusters, the standard deviation of the randomized data, or the number of randomized data points. I encourage the reader to change these variables and compare the k-Means results.

So, let's get started! Download and run **random_data.py**

The initial result should look something like this:



Also notice that above the graph, you get a comparison of the randomized centroids and the estimated k-Means centroids. If you look closely, for each cluster, the generated centroids are plotted as blue and the calculated centroids are red.



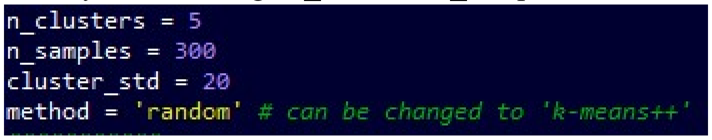
```
generated centroids
 [[23 86]
 [14 60]
 [90 19]
 [95 70]
 [97 14]]
```

```
estimated centroids
 [[  21.7   87.1]
 [  15.3   57.6]
 [  88.    21. ]
 [  95.    70. ]
 [ 104.    11.5]]
```

k-Means did a decent job predicting where the centroids are. When the standard deviation of the randomized clusters is increased, the estimated centroids will become less accurate.

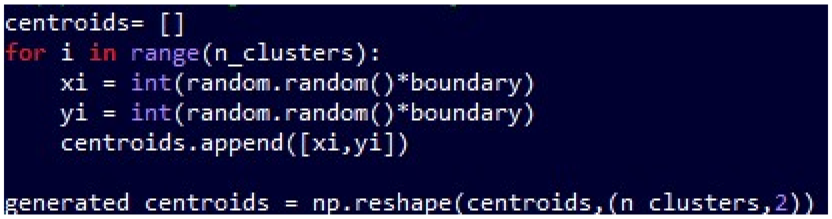
The code for this script is broken down into five main parts:

(1) Customizing input variables
Here, you can change n_clusters, n_samples, and cluster_std



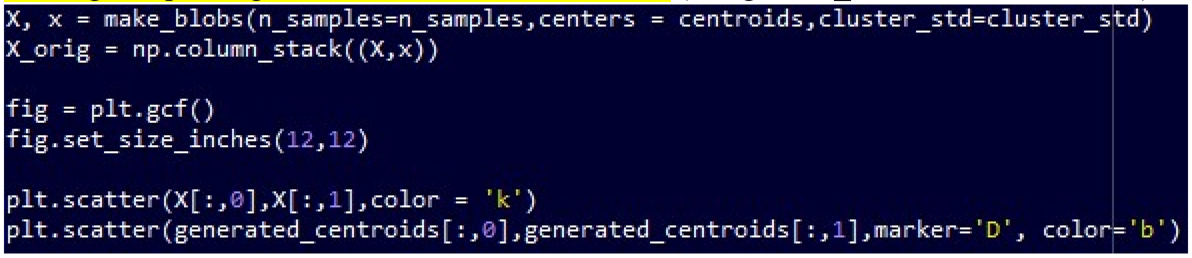
```
n_clusters = 5
n_samples = 300
cluster_std = 20
method = 'random' # can be changed to 'k-means++'
```

(2) Randomizing location of the clusters
A for loop creates random coordinates for the clusters to be centered around



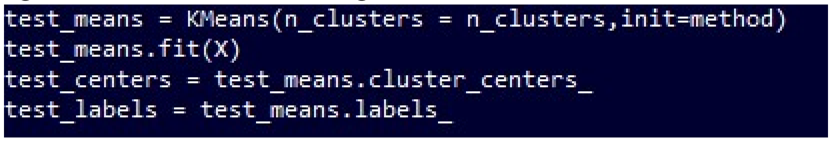
```
centroids= []
for i in range(n_clusters):
    xi = int(random.random()*boundary)
    yi = int(random.random()*boundary)
    centroids.append([xi,yi])

generated_centroids = np.reshape(centroids,(n_clusters,2))
```

(3) Making and plotting random blobs around centroids (using make_blobs from scikit-learn)



```
X, x = make_blobs(n_samples=n_samples,centers = centroids,cluster_std=cluster_std)
X_orig = np.column_stack((X,x))

fig = plt.gcf()
fig.set_size_inches(12,12)

plt.scatter(X[:,0],X[:,1],color = 'k')
plt.scatter(generated_centroids[:,0],generated_centroids[:,1],marker='D', color='b')
```

(4) Fitting randomized data using k-Means
scikit-learn's clustering library allows you to create a clustering object as a variable (KMeans in this case) and then call .fit() to apply on a data set. This notation will be used again for different clustering methods

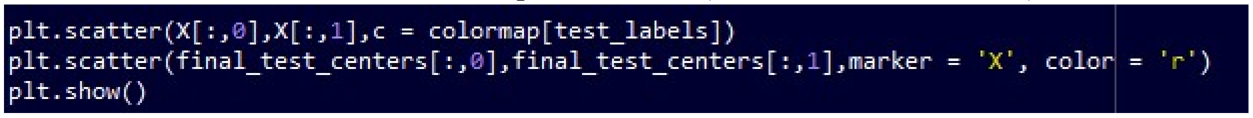


```
test_means = KMeans(n_clusters = n_clusters,init=method)
test_means.fit(X)
test_centers = test_means.cluster_centers_
test_labels = test_means.labels_
```

cluster_centers_ returns the calculated centroid coordinates. labels_ returns the label indices (in the range of [0,n_clusters-1]) to corresponding centroids. Note that .fit does not alter the data set X, but rather creates centroids and assigns each point in X to a centroid through labels_

(5) Plotting results

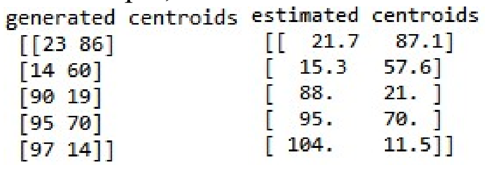
matplotlib.pyplot (plt) is used to create a colorized scatter plot of the k-Means results. The k-Means centroids are added to the plot in red Xs (marker = ‘X’, color = ‘r’)



```
plt.scatter(X[:,0],X[:,1],c = colormap[test_labels])
plt.scatter(final_test_centers[:,0],final_test_centers[:,1],marker = 'X', color = 'r')
plt.show()
```

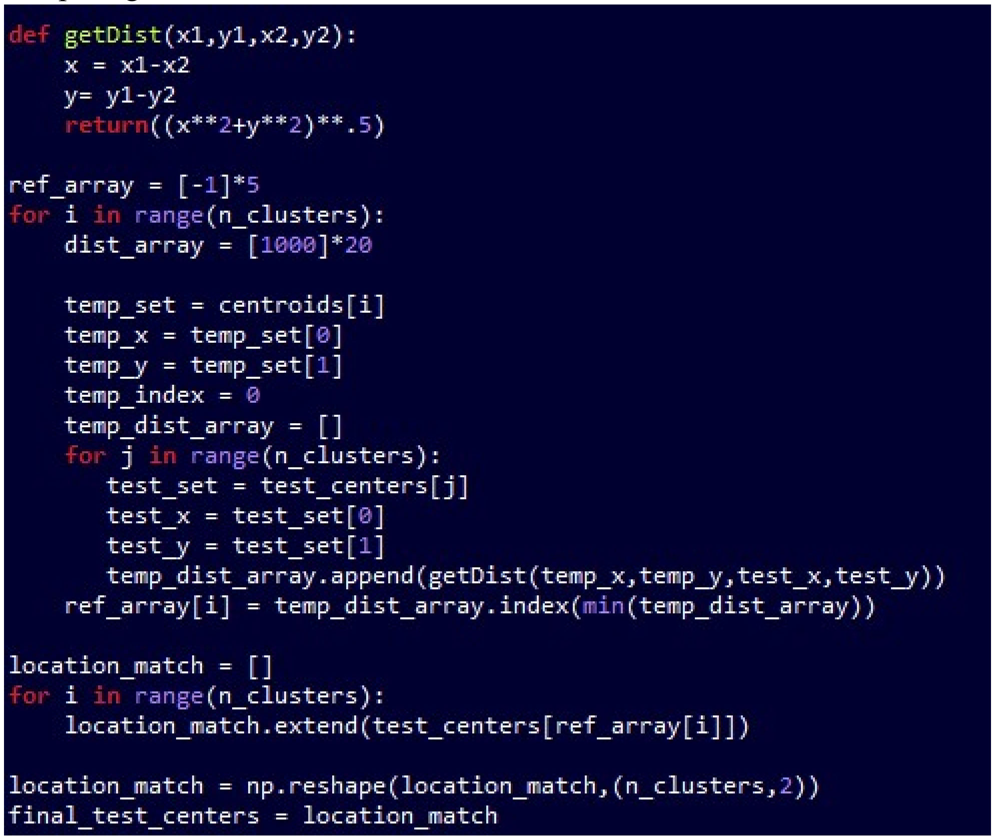
Remember that k-Means does not know where the initial random centroids are located. This means that it is possible to return the estimated centroids in a different order than their corresponding generated centroid.

For example, take the results from the initial run:



```
generated centroids estimated centroids
 [[23 86]           [[  21.7   87.1]
 [14 60]            [  15.3   57.6]
 [90 19]            [  88.    21. ]
 [95 70]            [  95.    70. ]
 [97 14]]           [ 104.    11.5]]
```

What happens if k-Means finds the [15.3, 57.6] centroid before the [21.7, 87.1] centroid? The centers need to be re-ordered before step (5). This is done with a double for loop comparing Euclidian distances between clusters



```
def getDist(x1,y1,x2,y2):
    x = x1-x2
    y= y1-y2
    return((x**2+y**2)**.5)

ref_array = [-1]*5
for i in range(n_clusters):
    dist_array = [1000]*20

    temp_set = centroids[i]
    temp_x = temp_set[0]
    temp_y = temp_set[1]
    temp_index = 0
    temp_dist_array = []
    for j in range(n_clusters):
       test_set = test_centers[j]
       test_x = test_set[0]
       test_y = test_set[1]
       temp_dist_array.append(getDist(temp_x,temp_y,test_x,test_y))
    ref_array[i] = temp_dist_array.index(min(temp_dist_array))

location_match = []
for i in range(n_clusters):
    location_match.extend(test_centers[ref_array[i]])

location_match = np.reshape(location_match,(n_clusters,2))
final_test_centers = location_match
```

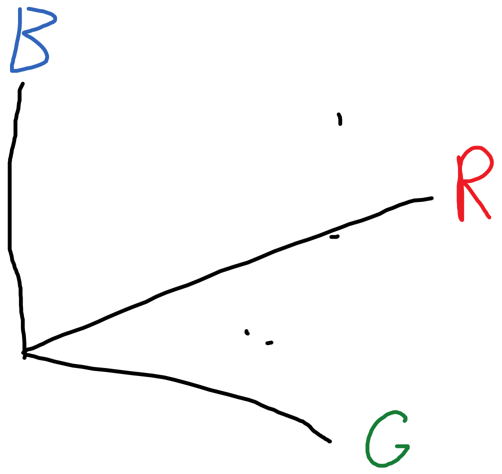
The concept to take from this very last block of code is how distance is used as a cost function when using k-Means. We are assuming that the corresponding randomized cluster to the estimated cluster is the one in the closest proximity.

k-Means aims to minimize the total distance that each point is away from its corresponding cluster. In another example, we will revisit this and see how distance can be used to compare k-Means fit against other methods.

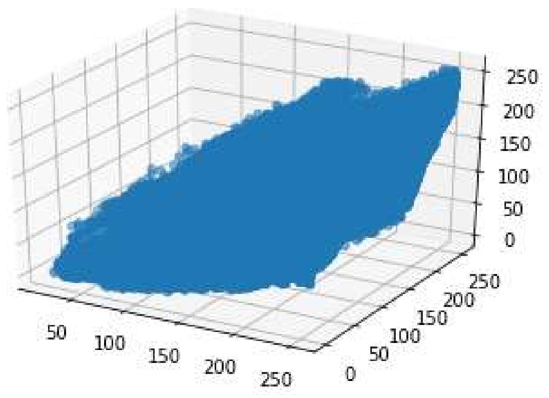
k-Means in higher dimensions

This first example demonstrated k-Means in a two dimensional plot. This does not mean that k-Means is limited to two dimensions. There are more abstract applications of k-Means as you apply it in higher dimensions. This next example will use k-Means in three dimensions.

Three dimensions can be used to show an object's location in space. It can also be used to represent color; along red, green, and blue axis like the figure below.



If indexed properly, points on a plot like this can represent an image. Below is a plot of all RGB values in a picture



A picture contain (Height x Width) pixels of RGB vales, so it is reasonable to assume it can be represented as an array of shape (Height, Width, 3). What would happen if we applied k-Means on something in three dimensions (a picture) then reconstructed the image?

The result would be a compressed image using only *k* colors.

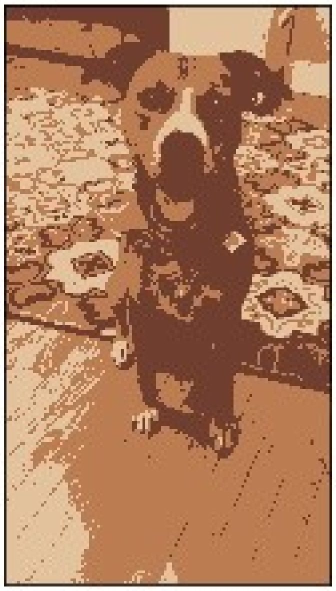
k-Means Imaging

For this example, you will need the following files from GitHub:

- **KMeansImage.py** (a Python library I created)
- **Kmeans_demo.py** (a demo script)
- **Civil.png and millie.jpg** (pictures used in the demo script)

Run the **kmeans_demo.py** script. This could take around one minute to run.

Your initial run should give an output with these generated images and arrays of corresponding RGB values.

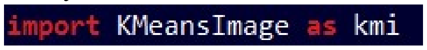


UIC
CIVIL AND
MATERIALS
ENGINEERING

By default, three colors (clusters) are used. See if you can figure out how to make it return five-color images.

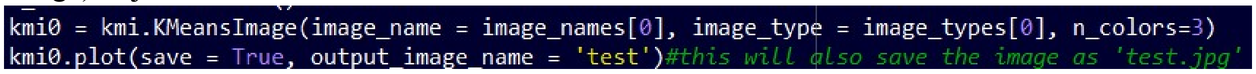
Notice that it outputs these images twice each. One set is done with functions and one set is done with objects. This will be explained further.

The code for this script is much shorter than the previous examples. This is because the library KMeansImage creates image objects wrapped in Scikit-learn's KMeans library. Much of the "dirty work" is done in the KMeansImage library file. It can be imported like so:



```
import KMeansImage as kmi
```

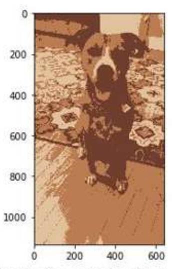
This allows the user to create the image and then plot (with the option to save the resulting image) in just two lines of code:



```
kmi0 = kmi.KMeansImage(image_name = image_names[0], image_type = image_types[0], n_colors=3)
kmi0.plot(save = True, output_image_name = 'test')#this will also save the image as 'test.jpg'
```

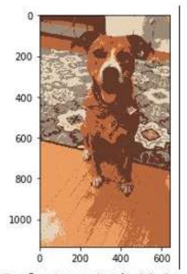
Here are the results of the same image in 3,5,7,9, and 16 clusters (colors) along with the runtime.
(

```
(1136, 640, 3)
```



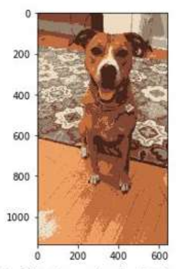
```
3 clusters took 15.5 seconds ( 0.26 minutes )
```

```
(1136, 640, 3)
```



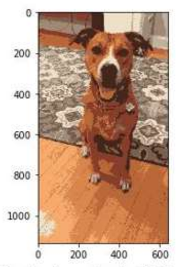
```
5 clusters took 26.2 seconds ( 0.44 minutes )
```

```
(1136, 640, 3)
```



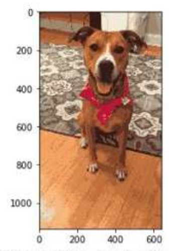
```
7 clusters took 32.6 seconds ( 0.54 minutes )
```

```
(1136, 640, 3)
```

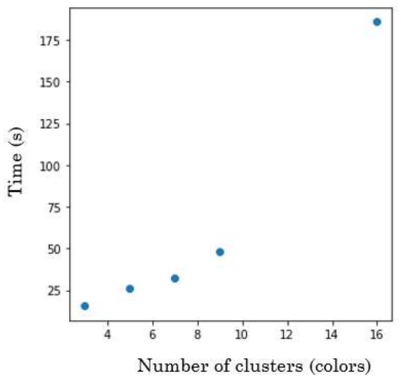


```
9 clusters took 48.0 seconds ( 0.8 minutes )
```

```
(1136, 640, 3)
```



```
16 clusters took 178.5 seconds ( 2.97 minutes )
(1136, 640, 3)
```



The runtime follows an exponential growth rate with respect to the number of clusters (colors).

…..

Density-based Clustering

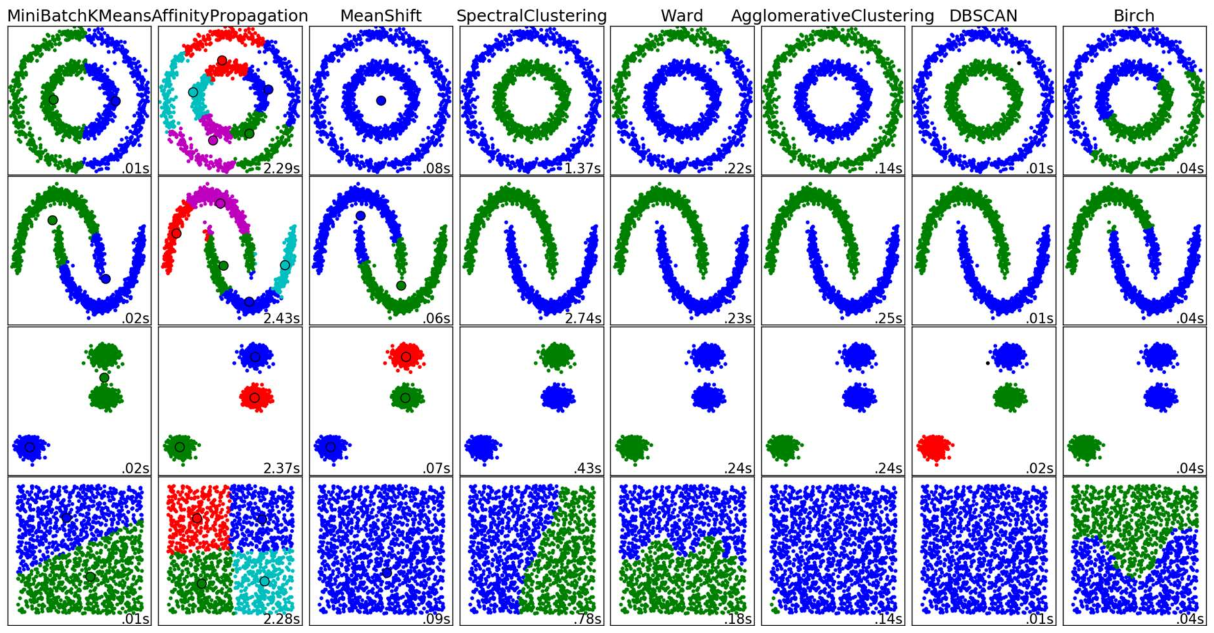
Another way of thinking about how to cluster a data set is the density, or proximity, of data point in relation to one another. The k-Means algorithm assigns a centroid to each cluster, but does not compute the distances between points within the cluster. This means that it works well with blob-like clusters (generated in the example) but does not always recognize other patterns. Density clustering takes a look at how close data points are to each other when creating clusters.

DBSCAN is the most common algorithm for density-based clustering. Scikit-learn has a built in class for DBSCAN in its cluster module. It has two main parameters:
(from scikit-learn.org)

- eps – (short for epsilon, ε)
  This is the maximum distance allowed for between two data points in the same cluster.
- min_samples
  The minimum number of data points required to become a cluster

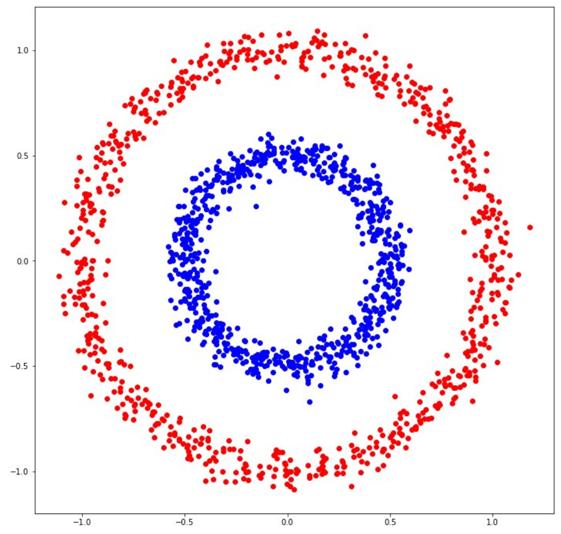
Note that these are listed as optional in the documentation, but rarely work with the default values given when applied to a dataset.

Also provided in scikit-learn is this infographic displaying results of varying clustering techniques on different shaped example datasets



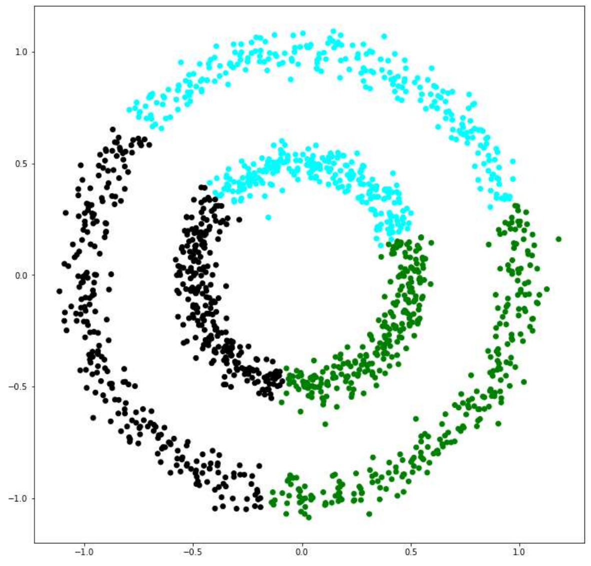
(*scikit-learn.org*)

Let’s apply DBSCAN to a simple (obvious) pattern. Scikit-learn’s samples generator can generate the enclosed double-circle shape in the top row of the previous figure
Download **density_scan.py** and run the file.
The code in this file takes this:

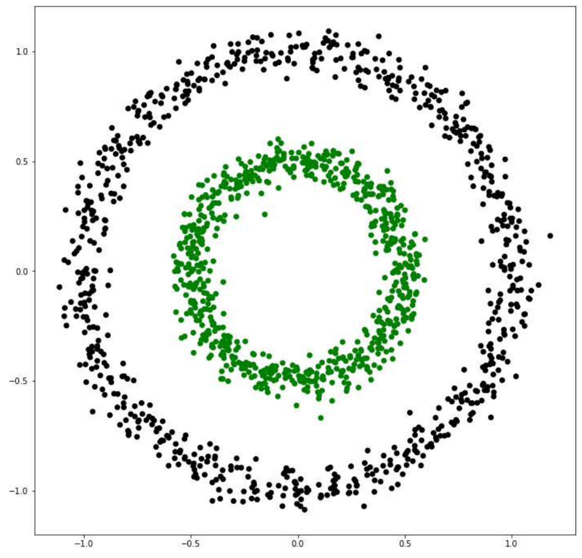


And applies both k-Means and DBSCAN

The results should look like this:



k-Means

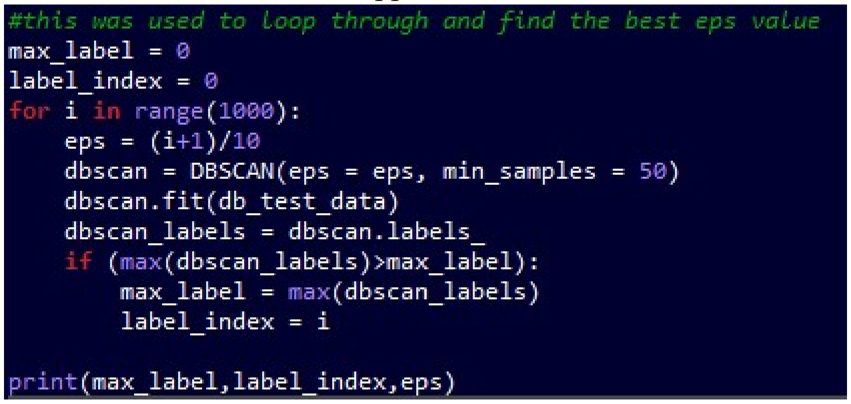


DBSCAN

DBSCAN recognizes shapes like circles and moons better than other clustering techniques but can often fail when used on data that doesn't follow a densely packed structure. When used in Python, there are two main reasons it can fail to identify clusters on a data set.

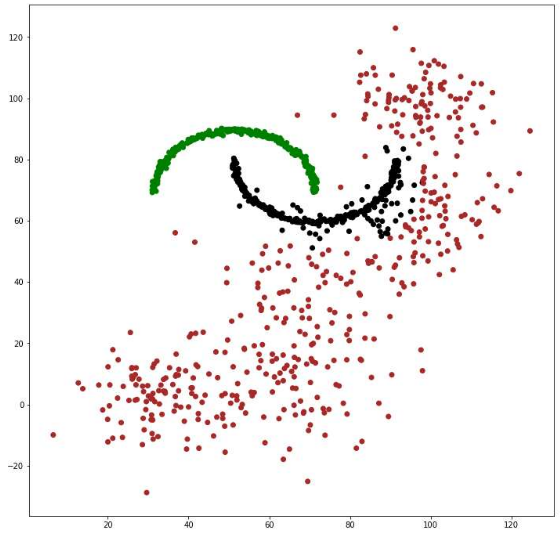
- Too much noise. If DBSCAN sees the data in a way where there is no clear cluster neighborhoods, it will essentially return zero clusters. This is indicated by an array of [-1, -1, -1,…] labels.
- All points are identified as one cluster. If DBSCAN returns an array of all zeros, [0,0,0,…] it means that all points are in one density cluster according to the parameters given.

If you encounter one of these two issues, it is likely that your parameters will have to be tweaked. Below is a code snippet of what was used to find the best eps (ε) value for the example:



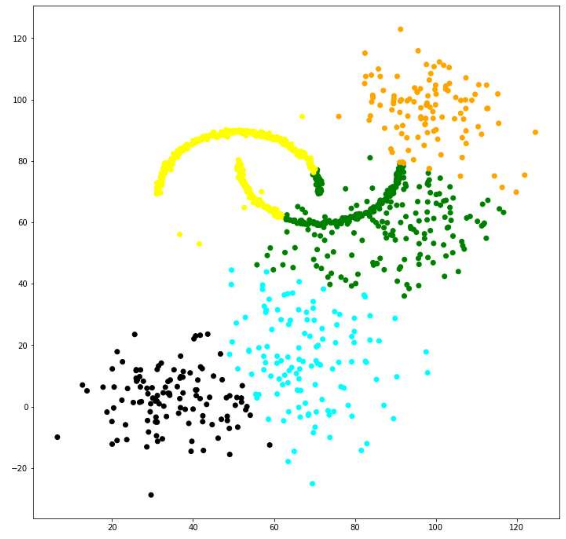
```
#this was used to loop through and find the best eps value
max_label = 0
label_index = 0
for i in range(1000):
    eps = (i+1)/10
    dbscan = DBSCAN(eps = eps, min_samples = 50)
    dbscan.fit(db_test_data)
    dbscan_labels = dbscan.labels_
    if (max(dbscan_labels)>max_label):
        max_label = max(dbscan_labels)
        label_index = i

print(max_label,label_index,eps)
```

Open the file called **randomized_dbscan.py**. By default, it generates two moon-shaped clusters and some blob-like clusters. See if you can edit the parameters to successfully identify the randomized shapes.
A successful run will look like this:



The two moons are clearly recognized with DBSCAN. Brown indicates noise [-1].

k-Means should return something like this, where the moons are not completely recognized as independent clusters.



If you are stuck, try using an eps of 5.5 and min_samples of 30 (with moon_samples still at 500)

Conclusion

Clustering is a powerful tool that can be used on a data set with no classifications and with little known information. It is not used to find an output result, e.g. $Y \rightarrow f(x)$, but rather to explore how the data can be grouped. Two main algorithms are k-Means and DBSCAN. Both are easily implemented in Python, so knowing the fundamental mathematics behind them is not as important as knowing how and when to use them.

k-Means is the best option when the data seems to be in clusters, but the clusters do not take on a specific pattern. It is designed to give a specified number of clusters minimizing the distance cost to the cluster centroid.

DBSCAN can easily detect circles, curves, and other irregular patterns in a data set. The drawback is that it is very difficult to set up, especially because the user cannot specify how many clusters to find. Common situations where DBSCAN clustering fails are when it detects too much noise or when it groups all data points into one cluster. This is typically fixable by changing its eps and min_samples parameters, but is tedious and time consuming.

Clustering does not classify its results, so even when an algorithm fits a set of data well, it cannot give a classification result to the data points. For this, classification algorithms are necessary.

Python (with scikit-learn) provides a platform where novice programmers can apply machine learning techniques and quickly get results, benefitting the scientific and computing communities.