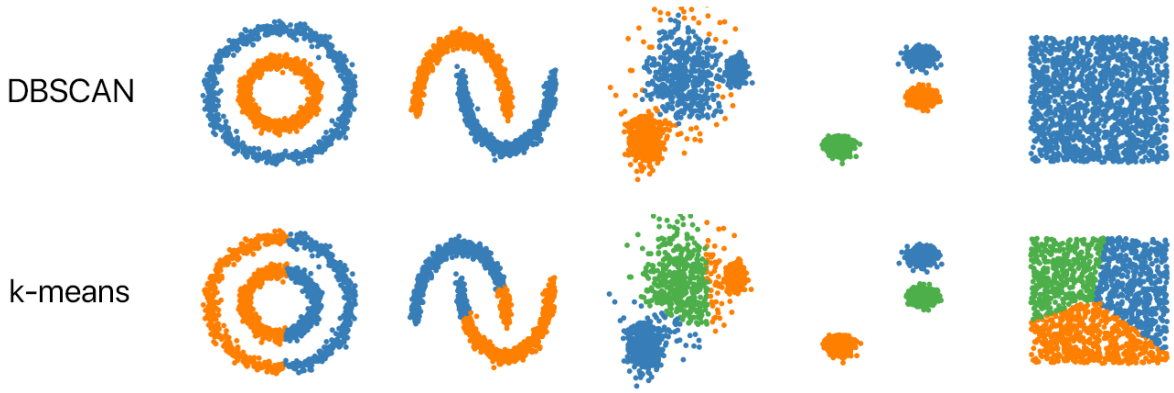
Density-Based Spatial Clustering of applications with noise (DBSCAN)

Why do we need a Density-Based clustering algorithm like DBSCAN when we already have K-means clustering?

- K-Means clustering may cluster loosely related observations together. Every observation becomes a part of some cluster eventually, even if the observations are scattered far away in the vector space.
- Since clusters depend on the mean value of cluster elements, each data point plays a role in forming the clusters. A slight change in data points might affect the clustering outcome.
- This problem is greatly reduced in DBSCAN due to the way clusters are formed. This is usually not a big problem unless we come across some odd shape data.
- Another challenge with k-means is that you need to specify the number of clusters ("k") in order to use it. Much of the time, we won't know what a reasonable k value is a priori.
- nice about DBSCAN is that you don't have to specify the number of clusters to use it. All you need is a function to calculate the distance between values and some guidance for what amount of distance is considered "close".
- DBSCAN also produces more reasonable results than k-means across a variety of different distributions. Below figure illustrates the fact:



Some definitions first:

Epsilon: This is also called eps. This is the distance till which we look for the neighbouring points.

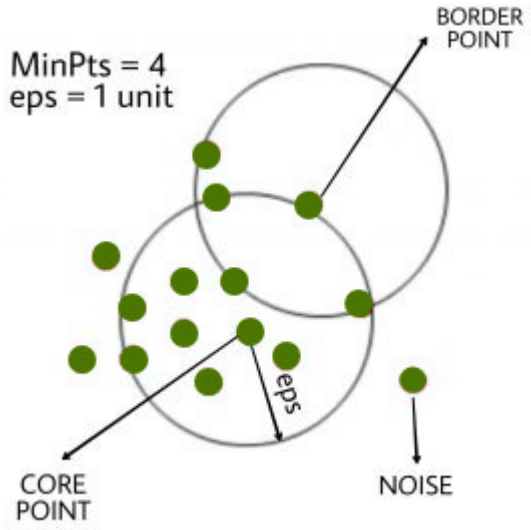
Min_points: The minimum number of points specified by the user.

Core Points: If the number of points inside the *eps radius* of a point is greater than or equal to the *min_points* then it's called a core point.

Border Points: If the number of points inside the *eps radius* of a point is less than the *min_points* and it lies within the *eps radius* region of a core point, it's called a border point.

Noise: A point which is neither a core nor a border point is a noise point.

Let's say if the eps=1 and min_points =4



Characteristics

- It identifies clusters of any shape in a data set, it means it can detect arbitrarily shaped clusters.
- It is based on intuitive notions of clusters and noise.
- It is very robust in detection of outliers in data set
- It requires only two points which are very insensitive to the order of occurrence of the points in data set

Algorithm Steps:

1. The algorithm starts with a random point in the dataset which has not been visited yet and its neighbouring points are identified based on the eps value.
2. If the point contains greater than or equal points than the min_pts, then the cluster formation starts and this point becomes a _core point_, else it's considered as noise. The thing to note here is that a point initially classified as noise can later become a border point if it's in the eps radius of a core point.
3. If the point is a core point, then all its neighbours become a part of the cluster. If the points in the neighbourhood turn out to be core points then their neighbours are also part of the cluster.
4. Repeat the steps above until all points are classified into different clusters or noises.

This algorithm works well if all the clusters are dense enough, and they are well separated by low-density regions.

Hands On Implementation And Compare All Three Algorithm

In [1]:
```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import matplotlib
```

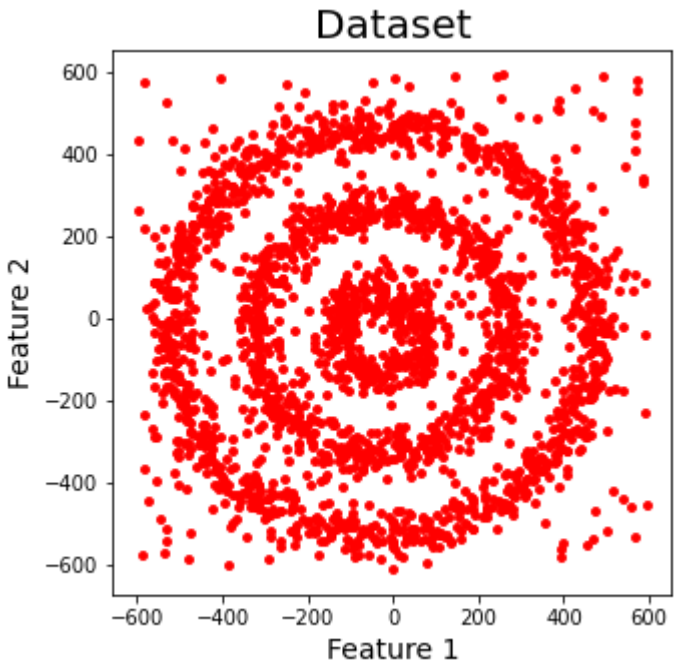
In [2]:
```
dataset = pd.read_csv('dbscan.csv')
df = dataset
df.head()
```

Out[2]:

	0	1
0	484.891555	-31.006357
1	489.391178	21.973916
2	462.886575	-27.599889
3	517.218479	5.588090
4	455.669049	1.982181

Let's plot these data points and see how they look in the feature space. Here, I use the scatter plot for plotting these data points. Use the following syntax:

In [3]:
```
plt.figure(figsize=(5,5))
plt.scatter(df['0'],df['1'],s=15,color='red')
plt.title('Dataset',fontsize=20)
plt.xlabel('Feature 1',fontsize=14)
plt.ylabel('Feature 2',fontsize=14)
plt.show()
```



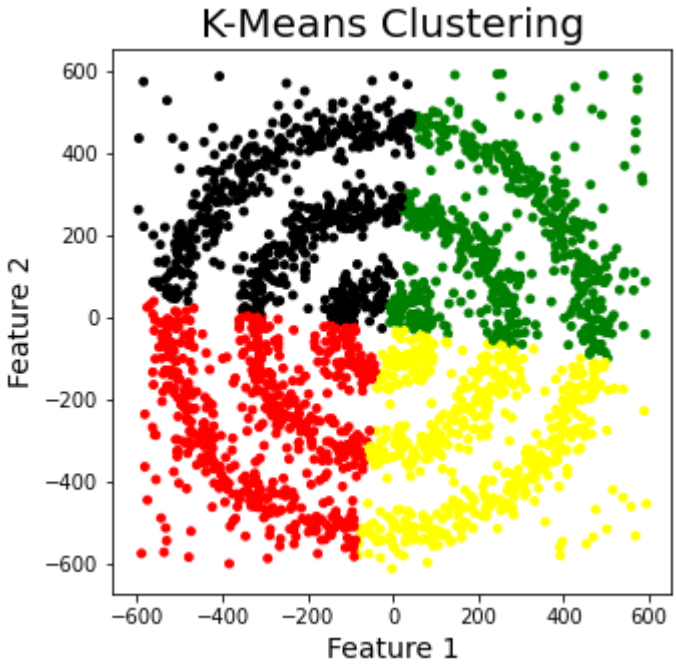
K-Means vs. Hierarchical vs. DBSCAN Clustering

K-Means

In [4]:
```
from sklearn.cluster import KMeans
k_means=KMeans(n_clusters=4,random_state=42)
k_means.fit(df[['0','1']])
```

Out[4]:
```
KMeans(n_clusters=4, random_state=42)
```

In [5]:
```
df['KMeans_labels']=k_means.labels_
colors=['red','green','yellow','black']
# Plotting resulting clusters
plt.figure(figsize=(5,5))
plt.scatter(df['0'],df['1'],c=df['KMeans_labels'],cmap=matplotlib.colors.ListedCo
plt.title('K-Means Clustering',fontsize=20)
plt.xlabel('Feature 1',fontsize=14)
plt.ylabel('Feature 2',fontsize=14)
plt.show()
```



- Here, K-means failed to cluster the data points into four clusters. Also, it didn't work well with noise. Therefore, it is time to try another popular clustering algorithm, i.e., Hierarchical Clustering.

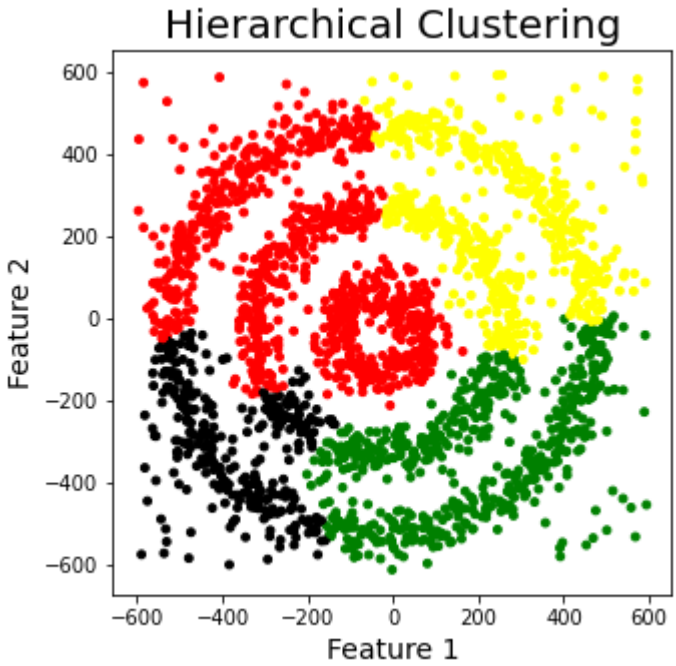
2. Hierarchical Clustering

In [6]:
```
from sklearn.cluster import AgglomerativeClustering
model = AgglomerativeClustering(n_clusters=4, affinity='euclidean')
model.fit(df[['0','1']])
```

Out[6]:
```
AgglomerativeClustering(n_clusters=4)
```

In [7]:
```
df['HR_labels']=model.labels_

# Plotting resulting clusters
plt.figure(figsize=(5,5))
plt.scatter(df['0'],df['1'],c=df['HR_labels'],cmap=matplotlib.colors.ListedColorm
plt.title('Hierarchical Clustering',fontsize=20)
plt.xlabel('Feature 1',fontsize=14)
plt.ylabel('Feature 2',fontsize=14)
plt.show()
```



hierarchical clustering algorithm also failed to cluster the data points properly. Now got to DBSCAN clustering

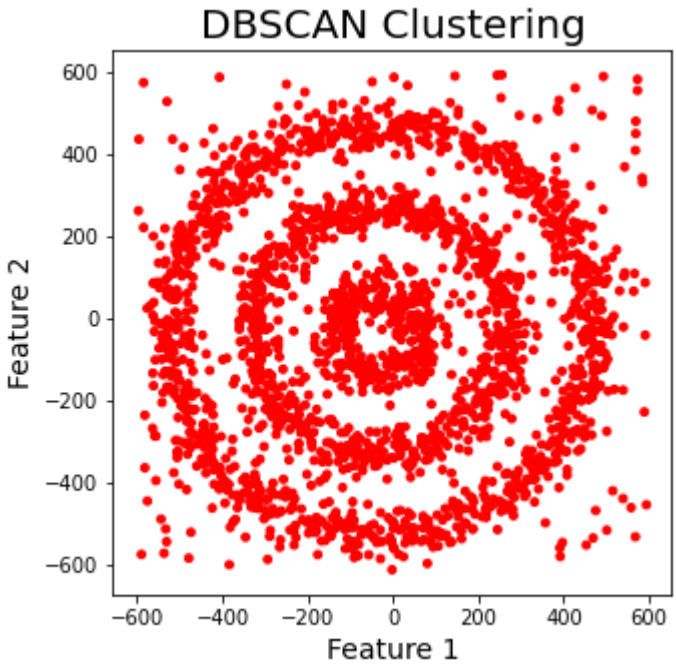
3. DBSCAN Clustering

In [8]:
```
from sklearn.cluster import DBSCAN
dbscan=DBSCAN()
dbscan.fit(df[['0','1']])
```

Out[8]:
```
DBSCAN()
```

In [9]:
```
df['DBSCAN_labels']=dbscan.labels_

# Plotting resulting clusters
plt.figure(figsize=(5,5))
plt.scatter(df['0'],df['1'],c=df['DBSCAN_labels'],cmap=matplotlib.colors.ListedCo
plt.title('DBSCAN Clustering',fontsize=20)
plt.xlabel('Feature 1',fontsize=14)
plt.ylabel('Feature 2',fontsize=14)
plt.show()
```

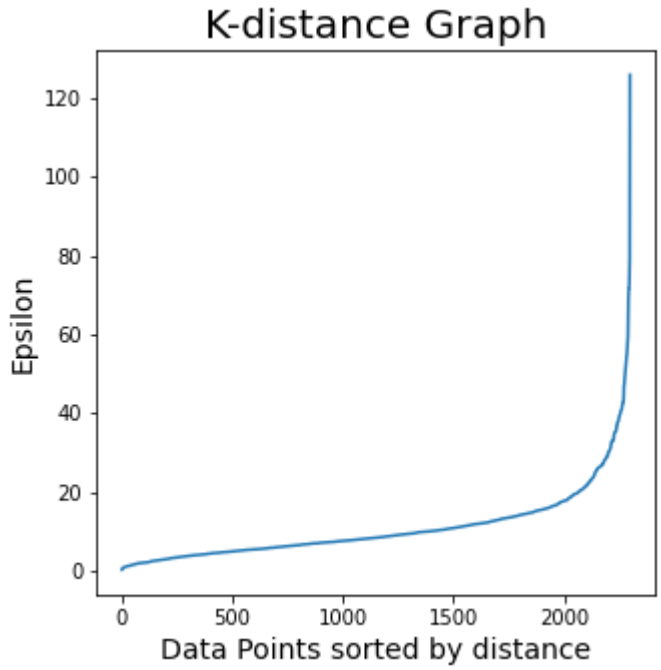


All the data points are now of red color which means they are treated as noise. It is because the value of epsilon is very small and we didn't optimize parameters. Therefore, we need to find the value of epsilon and minPoints and then train our model again.

For epsilon, I am using the K-distance graph. For plotting a K-distance Graph, we need the distance between a point and its nearest data point for all data points in the dataset. We obtain this using NearestNeighbors from sklearn.neighbors.

In [10]:
```
from sklearn.neighbors import NearestNeighbors
neigh = NearestNeighbors(n_neighbors=2)
nbrs = neigh.fit(df[['0','1']])
distances, indices = nbrs.kneighbors(df[['0','1']])
```

In [11]:
```
# Plotting K-distance Graph
distances = np.sort(distances, axis=0)
distances = distances[:,1]
plt.figure(figsize=(5,5))
plt.plot(distances)
plt.title('K-distance Graph',fontsize=20)
plt.xlabel('Data Points sorted by distance',fontsize=14)
plt.ylabel('Epsilon',fontsize=14)
plt.show()
```



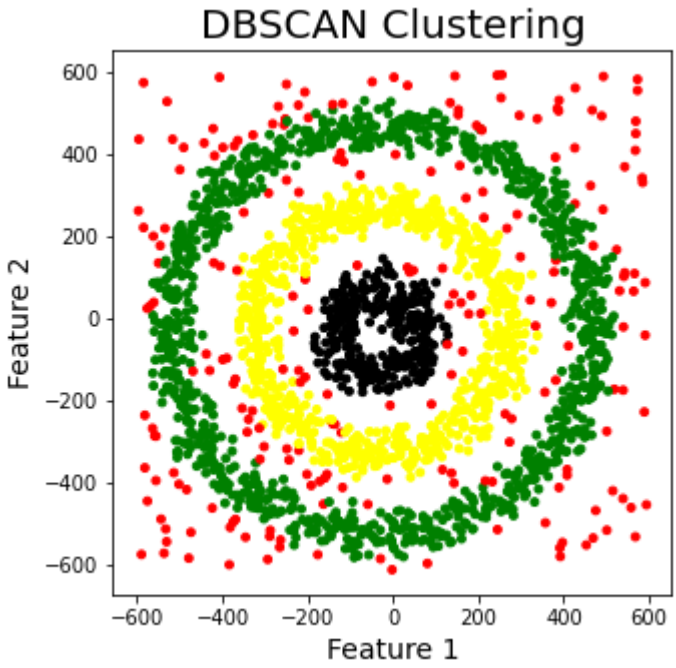
The optimum value of epsilon is at the point of maximum curvature in the K-Distance Graph, which is 30 in this case. Now, it's time to find the value of minPoints. The value of minPoints also depends on domain knowledge. This time I am taking minPoints as 6:

In [12]:
```
from sklearn.cluster import DBSCAN
dbscan_opt=DBSCAN(eps=30,min_samples=6)
dbscan_opt.fit(df[['0','1']])
```

Out[12]:
```
DBSCAN(eps=30, min_samples=6)
```

In [13]:
```
df['DBSCAN_opt_labels']=dbscan_opt.labels_

# Plotting the resulting clusters
plt.figure(figsize=(5,5))
plt.scatter(df['0'],df['1'],c=df['DBSCAN_opt_labels'],cmap=matplotlib.colors.List
plt.title('DBSCAN Clustering',fontsize=20)
plt.xlabel('Feature 1',fontsize=14)
plt.ylabel('Feature 2',fontsize=14)
plt.show()
```



DBSCAN amazingly clustered the data points into three clusters, and it also detected noise in the dataset represented by the red color.

One thing important to note here is that, though DBSCAN creates clusters based on varying densities, it struggles with clusters of similar densities. Also, as the dimension of data increases, it becomes difficult for DBSCAN to create clusters and it falls prey to the Curse of Dimensionality.

@@ Credit : https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/ (https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/)

In [14]:
```
df.head()
```

Out[14]:

	0	1	KMeans_labels	HR_labels	DBSCAN_labels	DBSCAN_opt_labels
0	484.891555	-31.006357	1	1	-1	0
1	489.391178	21.973916	1	2	-1	0
2	462.886575	-27.599889	1	1	-1	0
3	517.218479	5.588090	1	1	-1	0
4	455.669049	1.982181	1	2	-1	0