😊 K-means clustering

K-means clustering is a simple way to group similar things together in machine learning. Imagine you have a bunch of different items, and you want to organize them into groups based on their similarities. K-means helps you do that.

How it works:

1. **Start with some groups**: You tell the algorithm how many groups (or "clusters") you want to create. Let's say you want 3 groups.
2. **Pick starting points**: The algorithm randomly selects 3 points (centers) from the data. These points act as the "centers" of your clusters.
3. **Assign items to clusters**: Each item is compared to the centers, and the algorithm assigns it to the nearest center. So, if an item is closest to center A, it gets assigned to cluster A, and so on.
4. **Update the centers**: After all items are assigned to clusters, the algorithm recalculates the center of each cluster by averaging the positions of all the items in that cluster. This gives you new, more accurate centers.
5. **Repeat**: Steps 3 and 4 are repeated until the centers stop moving much. This means the groups are stable, and the algorithm has found the best way to organize the items.

Example:

Imagine you're a store owner and you have customer data, such as their ages and the amount they spend. You want to group your customers into 3 categories based on these two features (age and spending).

1. You tell K-means to divide your customers into 3 clusters.
2. It starts by picking 3 random "center" points in your data (maybe a 25-year-old who spends $100, a 40-year-old who spends $200, and a 60-year-old who spends $300).
3. Each customer is then assigned to the group of the nearest center.
4. The algorithm adjusts the center of each group by averaging the ages and spending of customers in that group.
5. It repeats this until it finds the best fit, so you end up with groups like:
    - **Group 1**: Young customers who spend less.
    - **Group 2**: Middle-aged customers with moderate spending.
    - **Group 3**: Older customers who spend the most.

This helps you understand the types of customers you have and potentially target them with personalized marketing.

Key Points:

- **"K"** is the number of groups you want (you choose this).

- It organizes items into groups based on their similarities.
- The algorithm keeps refining the groups to make them as accurate as possible.

K-means clustering is an **unsupervised learning algorithm** used to partition a dataset into distinct groups or clusters. The idea is to divide the data into *K* clusters, where each data point belongs to the cluster with the nearest mean (or centroid). The goal is to minimize the variance within each cluster while maximizing the variance between clusters.

Steps Involved in K-means Clustering:

1. **Choose the number of clusters (K)**: The algorithm requires you to specify the number of clusters you want to create in advance (this is why it's called "K-means").
2. **Initialize cluster centroids**: Randomly select *K* data points from the dataset as the initial centroids (the center points of the clusters).
3. **Assign each data point to the nearest centroid**: Each data point is assigned to the cluster whose centroid is the closest, based on the distance (usually Euclidean distance).
4. **Update centroids**: After assigning all the points to clusters, recalculate the centroids by finding the mean of all data points in each cluster.
5. **Repeat steps 3 and 4**: Reassign points to clusters and recalculate centroids until the centroids no longer move or until a maximum number of iterations is reached.
6. **Stop**: The algorithm stops when the centroids no longer change significantly or the number of iterations exceeds a threshold.

Example:

Let's take an example where we want to segment customers of an online store based on their age and annual spending:

- We have data on customers' **age** and **spending score** (how much they spend).
- We want to categorize these customers into 3 groups: low spenders, moderate spenders, and high spenders.

1. **Set K = 3**: We want to divide customers into 3 clusters.
2. **Initialize centroids**: The algorithm randomly picks 3 initial centroids from the dataset.
3. **Assign each customer to the nearest centroid**: Based on the distance between each customer's age and spending score, customers are assigned to one of the 3 clusters.
4. **Update centroids**: Once customers are assigned, the centroids of the clusters are recalculated as the mean of the customers' age and spending score in each cluster.
5. **Repeat**: The process repeats, with customers possibly being reassigned as centroids are updated, until the centroids stabilize and no further changes occur.

The final result will be three clusters of customers:

- Cluster 1 might consist of young, low-spending customers.

- Cluster 2 could consist of middle-aged, moderate spenders.
- Cluster 3 might be older, high-spending customers.

Objective of K-means:

- **Minimize intra-cluster variance** (make the points within a cluster as close to each other as possible).
- **Maximize inter-cluster variance** (make the clusters themselves as distinct as possible).

Key Concepts:

- **Centroid**: The center of a cluster, calculated as the mean of all the points in the cluster.
- **Distance Metric**: K-means usually uses **Euclidean distance** to measure the distance between points and centroids.

Pros:

- Simple to implement.
- Scales well to large datasets.
- Fast and efficient for finding distinct clusters in data.

Cons:

- You need to specify the number of clusters (K) in advance.
- It is sensitive to the initial placement of centroids (different starting points can lead to different results).
- Not suitable for non-spherical or complex-shaped clusters.
- It assumes clusters are of roughly equal size and density.

Applications:

- **Customer Segmentation**: Grouping customers based on purchasing behavior.
- **Image Compression**: Reducing the number of colors in an image.
- **Document Clustering**: Organizing a large set of documents into topic clusters.

In summary, K-means clustering is a powerful tool for finding natural groupings in data, but it requires careful tuning (like choosing the right value of K) and may not work well if the data isn't well-suited to spherical clusters.

😄 To begin with the Lab:

1. In your Jupyter Notebook, first you need to upload the Wholesale Customers Data CSV file, then you need to create a new Python Kernel.
2. Then you need to import the important libraries and load your dataset as a data frame and display few entries from it.

jupyter KMeans & Hierarchial Clustering Last Checkpoint: a minute ago (unsaved changes)

Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 (ipykernel)

Run Code

Clustering

K-Means Clustering

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

In [2]:
```
data = pd.read_csv('Wholesale customers data.csv')
data.head()
```

Out[2]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
0	2	3	12669	9656	7561	214	2674	1338
1	2	3	7057	9810	9568	1762	3293	1776
2	2	3	6353	8808	7684	2405	3516	7844
3	1	3	13265	1196	4221	6404	507	1788
4	2	3	22615	5410	7198	3915	1777	5185

3. Using the below command we have described our dataset as you can see below. By looking at the data we can say that there is a lot of variation in the magnitude of the data.

In [3]:
```
data.describe()
```

Out[3]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
count	440.000000	440.000000	440.000000	440.000000	440.000000	440.000000	440.000000	440.000000
mean	1.322727	2.543182	12000.297727	5796.265909	7951.277273	3071.931818	2881.493182	1524.870455
std	0.468052	0.774272	12647.328865	7380.377175	9503.162829	4854.673333	4767.854448	2820.105937
min	1.000000	1.000000	3.000000	55.000000	3.000000	25.000000	3.000000	3.000000
25%	1.000000	2.000000	3127.750000	1533.000000	2153.000000	742.250000	256.750000	408.250000
50%	1.000000	3.000000	8504.000000	3627.000000	4755.500000	1526.000000	816.500000	965.500000
75%	2.000000	3.000000	16933.750000	7190.250000	10655.750000	3554.250000	3922.000000	1820.250000
max	2.000000	3.000000	112151.000000	73498.000000	92780.000000	60869.000000	40827.000000	47943.000000

4. So, now we will bring all the variables to the same magnitude. After performing the scaling activity, we can see that all the columns in my dataset now have a mean of zero and a standard deviation of one. This process has standardized the dataset, ensuring that all columns share the same mean and standard deviation values.

In [4]:
```
from sklearn.preprocessing import StandardScaler
ss = StandardScaler()
data_scaled = ss.fit_transform(data)

data_scaled = pd.DataFrame(data_scaled,columns=data.columns)
data_scaled.describe()
```

Out[4]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen
count	4.400000e+02	4.400000e+02	4.400000e+02	440.000000	4.400000e+02	4.400000e+02	4.400000e+02	4.400000e+02
mean	1.614870e-17	3.552714e-16	-3.431598e-17	0.000000	-4.037175e-17	3.633457e-17	2.422305e-17	-8.074349e-18
std	1.001138e+00	1.001138e+00	1.001138e+00	1.001138	1.001138e+00	1.001138e+00	1.001138e+00	1.001138e+00
min	-6.902971e-01	-1.995342e+00	-9.496831e-01	-0.778795	-8.373344e-01	-6.283430e-01	-6.044165e-01	-5.402644e-01
25%	-6.902971e-01	-7.023369e-01	-7.023339e-01	-0.578306	-6.108364e-01	-4.804306e-01	-5.511349e-01	-3.964005e-01
50%	-6.902971e-01	5.906683e-01	-2.767602e-01	-0.294258	-3.366684e-01	-3.188045e-01	-4.336004e-01	-1.985766e-01
75%	1.448652e+00	5.906683e-01	3.905226e-01	0.189092	2.849105e-01	9.946441e-02	2.184822e-01	1.048598e-01
max	1.448652e+00	5.906683e-01	7.927738e+00	9.183650	8.936528e+00	1.191900e+01	7.967672e+00	1.647845e+01

5. Now that we have scaled the data, we can now go ahead and apply this K-means clustering algorithm.

6. The code is using the KMeans clustering algorithm to group data into clusters. First, it initializes the algorithm to create two clusters and fits it to the scaled data. Then, it calculates the inertia, which measures how tightly the clusters are formed.
7. Next, an Elbow Plot is created to help determine the optimal number of clusters. It does this by running the KMeans algorithm for a range of cluster counts from 2 to 19, recording the inertia for each. Finally, it plots the number of clusters against the inertia values, allowing you to visually assess where adding more clusters no longer significantly improves the model.

```
In [5]: from sklearn.cluster import KMeans
        km = KMeans(n_clusters=2)
        km.fit(data_scaled)
```

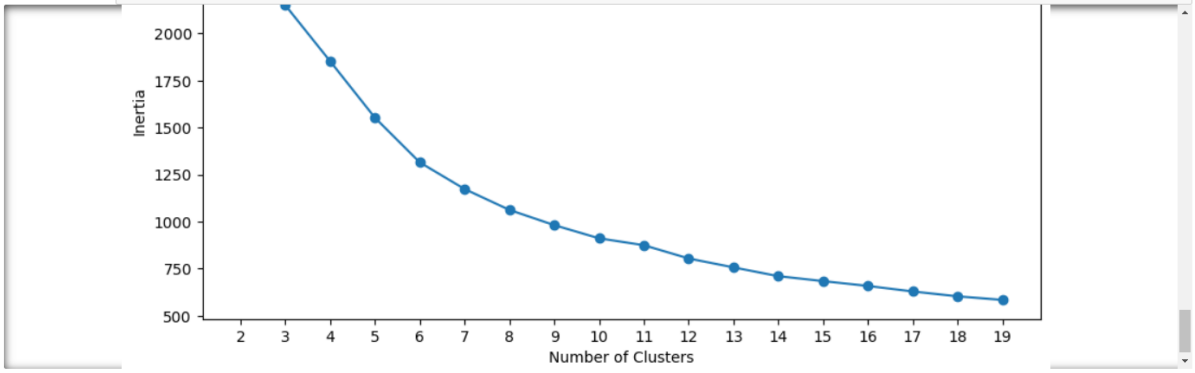
```
C:\Users\PULKIT\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will c
hange from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\PULKIT\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1382: UserWarning: KMeans is known to have a memory leak
on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OM
P_NUM_THREADS=2.
  warnings.warn(
```

```
Out[5]: KMeans
        KMeans(n_clusters=2)
```

```
In [6]: km.inertia_
Out[6]: 2599.38555935614
```

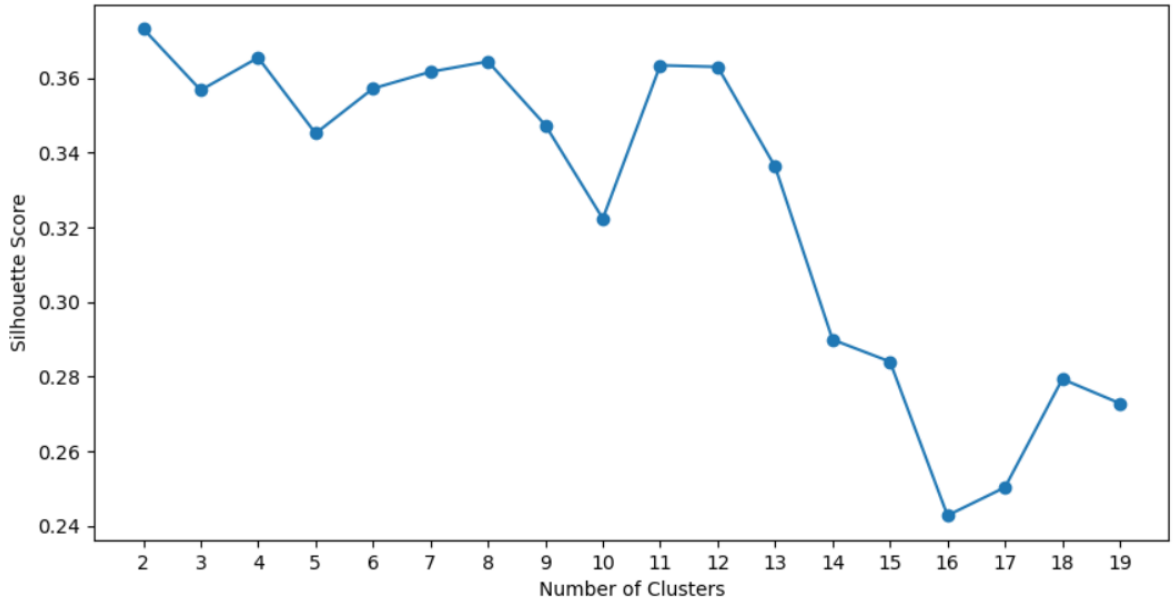

```
In [6]: km.inertia_
Out[6]: 2599.38555935614
```


```
In [7]: # Elbow Plot
        sse = []
        for cluster in range(2,20):
            km = KMeans(n_clusters=cluster)
            km.fit(data_scaled)
            sse.append(km.inertia_)

        out = pd.DataFrame({'Cluster':range(2,20), 'SSE': sse})
        plt.figure(figsize=(10,5))
        plt.plot(out['Cluster'], out['SSE'], marker='o')
        plt.xlabel("Number of Clusters")
        plt.xticks(range(2,20))
        plt.ylabel("Inertia")
        plt.show()
```



8. This code is calculating the Silhouette Score to evaluate the quality of clusters created by the KMeans algorithm. It runs the algorithm for different numbers of clusters, from 2 to 19, and for each configuration, it computes the Silhouette Score, which measures how similar an object is to its own cluster compared to other clusters.
9. The results are stored in a DataFrame, which is then used to create a plot showing the relationship between the number of clusters and the Silhouette Score. This plot helps visualize which number of clusters provides the best separation and cohesion, aiding in selecting the optimal number of clusters for the data.

In [8]:
```
# Silhoutte Score
# Elbow Plot
from sklearn.metrics import silhouette_score
score = []
for cluster in range(2,20):
    km = KMeans(n_clusters=cluster)
    km.fit(data_scaled)
    score.append(silhouette_score(data_scaled,km.labels_))

out = pd.DataFrame({'Cluster':range(2,20), 'Score': score})
plt.figure(figsize=(10,5))
plt.plot(out['Cluster'], out['Score'], marker='o')
plt.xlabel("Number of Clusters")
plt.xticks(range(2,20))
plt.ylabel("Silhouette Score")
plt.show()
```



10. In this code, the KMeans algorithm is set up to create 7 clusters, and it is applied to the scaled data. After fitting the model, each data point is assigned a cluster label, which is stored in a new column called 'cluster' in the original data.
11. The code then counts how many data points belong to each cluster and displays this information. Finally, it checks the shape of the data, which indicates the number of rows and columns, confirming the dataset's dimensions after adding the cluster labels.

In [9]:
```
km = KMeans(n_clusters=7)
km.fit(data_scaled)
data['cluster'] = km.labels_
data['cluster'].value_counts()
```

```
C:\Users\PULKIT\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:870: FutureWarning: The default value of `n_init` will c
hange from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  warnings.warn(
C:\Users\PULKIT\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1382: UserWarning: KMeans is known to have a memory leak
on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OM
P_NUM_THREADS=2.
  warnings.warn(
```

Out[9]:
```
cluster
1    201
2    124
4     60
0     41
3     11
5      2
6      1
Name: count, dtype: int64
```

In [10]:
```
data.shape
```

Out[10]:
```
(440, 9)
```

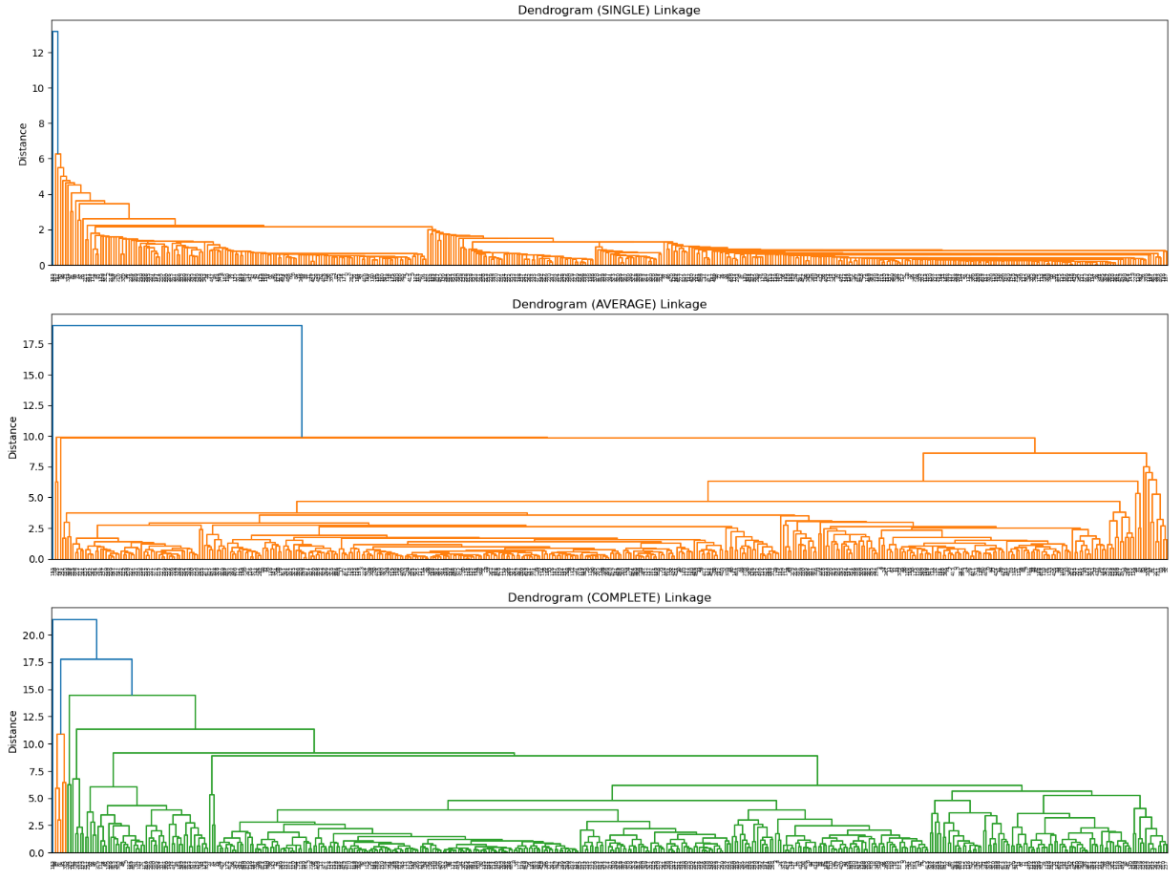
Hierarchical clustering is a method of grouping data into clusters based on their similarities, creating a tree-like structure called a **dendrogram**. It works by either:

- **Agglomerative (bottom-up)**: Starting with each data point as its own cluster, then gradually merging the closest clusters until all points belong to one big cluster.
- **Divisive (top-down)**: Starting with one large cluster, then splitting it into smaller clusters until each point is its own cluster.

This method allows you to see different levels of grouping, from smaller clusters to larger ones, and is useful for visualizing how data naturally forms groups.

12. This code generates dendrograms for hierarchical clustering using three different linkage methods: single, average, and complete.
13. **Linkage Methods**: These methods determine how the distance between clusters is calculated.
    a. **Single linkage** measures the shortest distance between points in different clusters.
    b. **Average linkage** calculates the average distance between all pairs of points in different clusters.
    c. **Complete linkage** uses the longest distance between points in different clusters.
14. **Creating Dendrograms**: For each method, the code computes the hierarchical clustering using the `linkage` function on the scaled data, then creates a dendrogram to visualize the results.
15. **Plotting**: Each dendrogram is plotted in its own subplot, labeled with the corresponding linkage method and showing the distance between clusters on the y-axis. This visualization helps in understanding the relationships and distances between the clusters formed.

In [11]:
```
from scipy.cluster.hierarchy import dendrogram,linkage

methods = ['single','average','complete']

fig,axes = plt.subplots(len(methods), 1, figsize=(20,15))

for i,method in enumerate(methods):
    z = linkage(data_scaled, metric='euclidean', method=method)
    dendrogram(z,ax=axes[i]);
    axes[i].set_title(f"Dendrogram ({method.upper()}) Linkage")
    axes[i].set_ylabel('Distance')
```



16. Now with this Dendrogram structure, what I'm going to do right now is I'll go ahead and perform the agglomerative clustering.

17. This code applies Agglomerative Clustering, a hierarchical clustering method, to the scaled data.

    - **Initialization**: The algorithm is set to create 4 clusters, using Euclidean distance to measure distances between data points.
    - **Linkage Method**: It employs the complete linkage method, which considers the maximum distance between points in different clusters when forming clusters.
    - **Fitting the Model**: The model is then fit to the scaled data, meaning it assigns each data point to one of the 4 clusters based on the hierarchical clustering approach. This allows for the exploration of cluster structure in the dataset.

In [12]:
```
from sklearn.cluster import AgglomerativeClustering
hierarchial = AgglomerativeClustering(n_clusters=4, affinity='euclidean',
                                      linkage='complete')
hierarchial.fit(data_scaled)
```

```
C:\Users\PULKIT\anaconda3\Lib\site-packages\sklearn\cluster\_agglomerative.py:983: FutureWarning: Attribute `affinity` was depr
ecated in version 1.2 and will be removed in 1.4. Use `metric` instead
  warnings.warn(
```

Out[12]:
```
AgglomerativeClustering(affinity='euclidean', linkage='complete', n_clusters=4)
```

18. This code adds the cluster labels from the Agglomerative Clustering model to the original data. A new column called 'hierarcy' is created, which contains the cluster assignment for each data point.

19. Finally, data.head() displays the first few rows of the updated dataset, allowing you to see the original data along with the newly added 'hierarcy' column that indicates which cluster each data point belongs to.

```
In [13]: data['hierarcy'] = hierarchial.labels_
         data.head()
```

Out[13]:

	Channel	Region	Fresh	Milk	Grocery	Frozen	Detergents_Paper	Delicassen	cluster	hierarcy
0	2	3	12669	9656	7561	214	2674	1338	2	0
1	2	3	7057	9810	9568	1762	3293	1776	2	0
2	2	3	6353	8808	7684	2405	3516	7844	2	0
3	1	3	13265	1196	4221	6404	507	1788	1	0
4	2	3	22615	5410	7198	3915	1777	5185	2	0

```
In [14]: data['hierarcy'].value_counts()

Out[14]: hierarcy
         0    432
         2      5
         1      2
         3      1
         Name: count, dtype: int64
```

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm that groups data points based on how close they are to each other. It identifies dense regions of data (clusters) and separates them from less dense areas (noise). Unlike K-means, you don't need to specify the number of clusters in advance.

DBSCAN works by:

- Finding "core points" that have many nearby neighbors.
- Expanding clusters around these core points by connecting neighboring points.
- Ignoring points in low-density areas as noise.

It's great for discovering clusters of various shapes and handling noise or outliers in the data.

DB Scan

Density Based Spatial Clustering of Applications with Noise

In [16]:
```
from sklearn.cluster import DBSCAN
dbs = DBSCAN(eps=1)
data['DBScan'] = dbs.fit_predict(data_scaled)
data['DBScan'].value_counts()
```

Out[16]:
```
DBScan
 1    187
 0     83
-1     81
 2     50
 4     26
 5      7
 3      6
Name: count, dtype: int64
```