

# Clustering

## Team Members:

- 1- Abdullah Abdelhakeem
- 2- Mohamed Mostafa
- 3- ElShaimaa Hassan
- 4- Mohamed Sebaie
- 5- Osama Ahmed
- 6- Lamiaa Omar

**Supervisor:** Dr/Shahira





**Introduction 01**

**K-Means Algorithm 02**

**Mean-Shift Algorithm 03**

**K-Means/Mean-Shift Merging 04**

**DBSCAN Algorithm 05**

**Conclusion 06**



# Clustering

Clustering/Cluster analysis is an unsupervised machine learning task.

It involves automatically discovering natural grouping in data. Unlike supervised learning (like predictive modeling), clustering algorithms only interpret the input data and find natural groups or clusters in feature space.

# Types of Clustering

## Clustering

### Partitioning

- K Means
- K Medoids
- K Modes

### Hierarchical

- Agglomerative (Bottom-Up)
- Divisive (Top-Down)

### Density-Based

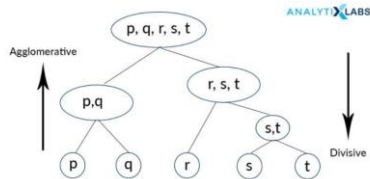
- DBSCAN
- HDBSCAN
- Mean Shift

### Model-Based

- EM
- COBWEB
- CLASSIT

### Grid-Based

- Sting
- WaveCluster






**K-Means**



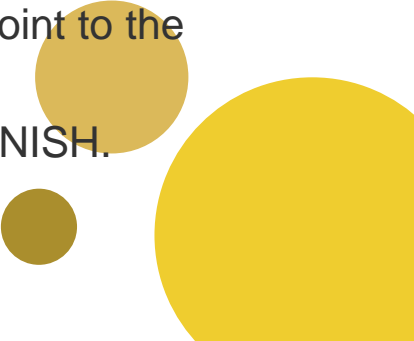
# K-Means Definition

K-means clustering is a method of vector quantization, that aims to partition  $n$  observations into  $k$  clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

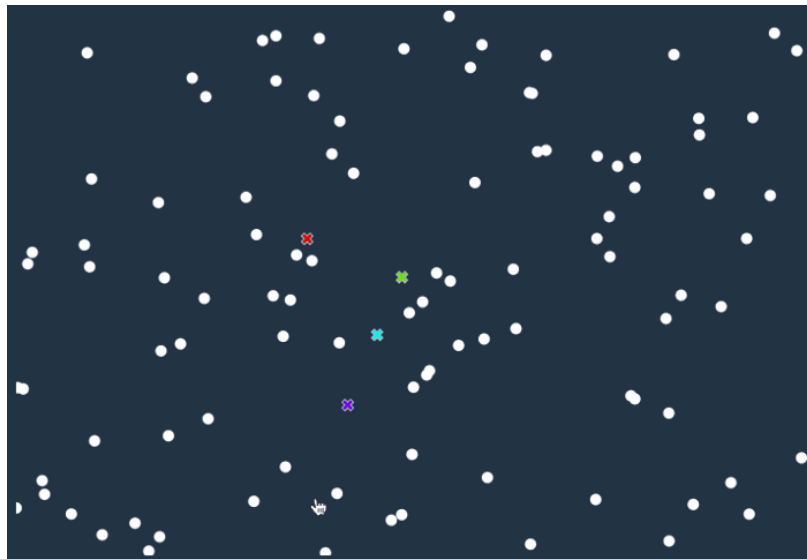




# K-Means Steps

- **Step-1:** Select the number  $K$  to decide the number of clusters.
  - **Step-2:** Select random  $K$  points or centroids. (It can be other from the input dataset).
  - **Step-3:** Assign each data point to their closest centroid, which will form the predefined  $K$  clusters.
  - **Step-4:** Calculate the variance and place a new centroid of each cluster.
  - **Step-5:** Repeat the third steps, which means re-assign each datapoint to the new closest centroid of each cluster.
  - **Step-6:** If any reassignment occurs, then go to step-4 else go to FINISH.
  - **Step-7:** The model is ready.
- 

# K-Means Animations





# K-Means Code

```
class K_Means:
    def __init__(self, k=2, tol=0.001, max_iter=300):
        self.k = k
        self.tol = tol
        self.max_iter = max_iter

    def fit(self, data):

        # centroids : contains the updated centroids
        # set the first k data points as centroids
        self.centroids = {}
        for i in range(self.k):
            self.centroids[i] = data[i]

        for i in range(self.max_iter):

            # classifications : carry the mapping of each centroid index
            # and the nearest points to each
            # initialize classifications dictionary with empty lists
            self.classifications = {}
            for i in range(self.k):
                self.classifications[i] = []

            # group points to their nearest centroid based on L2 norm
            for featureset in data:
                distances = [np.linalg.norm(featureset-self.centroids[centroid])
                             for centroid in self.centroids]
                classification = distances.index(min(distances))
                self.classifications[classification].append(featureset)
```

```
        # save centroids before updates to terminate when no change occur
        prev_centroids = dict(self.centroids)

        # update the centroids by averaging each collected group of points
        for classification in self.classifications:
            self.centroids[classification] = np.average(self.classifications[classification],axis=0)

        # terminate when the change rate is smaller than the tolerance
        # ( the loop also terminates when it reaches the max number of iterations)
        optimized = True

        for c in self.centroids:
            original_centroid = prev_centroids[c]
            current_centroid = self.centroids[c]
            if np.sum((current_centroid-original_centroid)/original_centroid*100.0) > self.tol:
                optimized = False

        if optimized:
            break

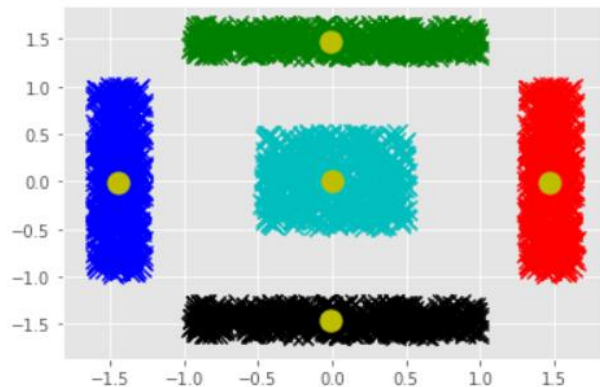
    return self.classifications, self.centroids
```

# K-Means

## Results ( K = 5 & K = 10 )

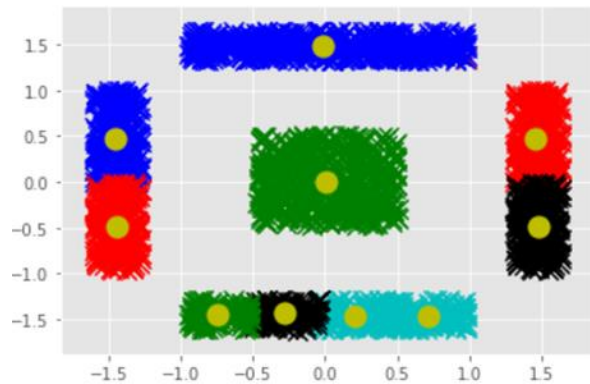
```
1 # perform K-means on dataset
2 tic = time.perf_counter()
3 kmeans = K_Means(k = 5)
4 classifications, centroids = kmeans.fit(Z)
5 toc = time.perf_counter()
6 print(f"K-means took {toc - tic:0.4f} seconds")
```

K-means took 0.9140 seconds



```
1 # perform K-means on dataset
2 tic = time.perf_counter()
3 kmeans = K_Means(k = 10)
4 classifications, centroids = kmeans.fit(Z)
5 toc = time.perf_counter()
6 print(f"K-means took {toc - tic:0.4f} seconds")
```

K-means took 2.3774 seconds





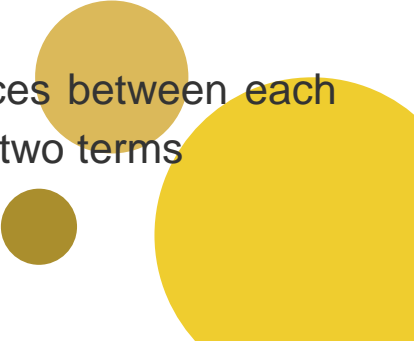
# How to Find Optimum Value of K ?

There are some different methods to choose k value ,here we focus on elbow method:

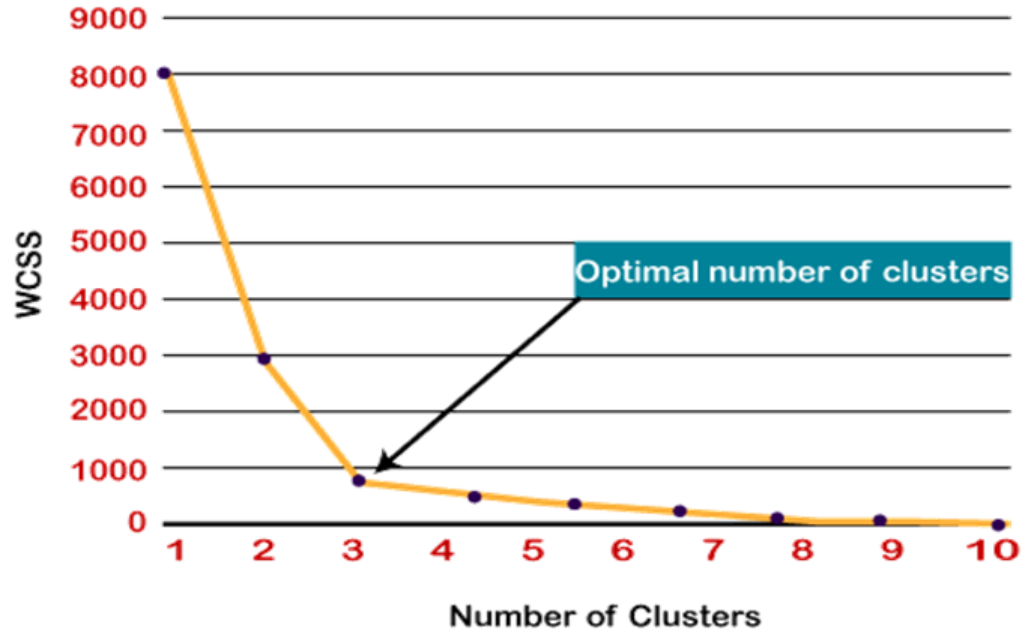
- This method uses the concept of **WCSS** ( Within Cluster Sum of Squares ) which defines the total variations within a cluster.

$$WCSS = \sum_{P_i \text{ in Cluster1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster3}} \text{distance}(P_i, C_3)^2$$

$\sum_{P_i \text{ in Cluster1}} \text{distance}(P_i, C_1)^2$ : is the sum of the square of the distances between each data point and its centroid within a cluster1 and the same for the other two terms

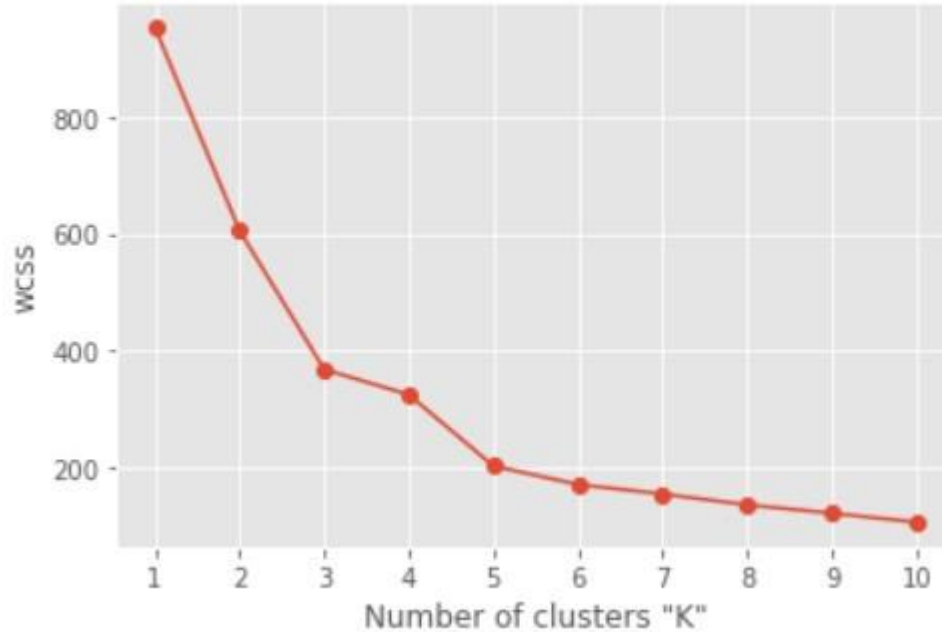


# Elbow Method



# Elbow Method on dataset

K-means executed in 10.2701 seconds



**Ideal K value =5**

# K-Means

## Code Analysis


```
def fit(self, data):
    self.centroids = {}
    for i in range(self.k): -----> O(K)
        self.centroids[i] = data[i]
    for i in range(self.max_iter):
        self.classifications = {}
        for i in range(self.k): -----> O(i * k)
            self.classifications[i] = []
        for featureset in data:
            distances = [np.linalg.norm(featureset - self.centroids[centroid]) -----> O(i * k * n)
                          for centroid in self.centroids]
            classification = distances.index(min(distances))
            self.classifications[classification].append(featureset)
        prev_centroids = dict(self.centroids)
        for classification in self.classifications: -----> O(i * k)
            self.centroids[classification] = np.average(self.classifications[classification], axis=0)
        optimized = True
        for c in self.centroids: -----> O(i * k)
            original_centroid = prev_centroids[c]
            current_centroid = self.centroids[c]
            if np.sum((current_centroid - original_centroid) / original_centroid * 100.0) > self.tol:
                optimized = False
    if optimized:
        break
    return self.classifications, self.centroids
```



# K-Means Code Analysis


Assuming small features for the data

```
for i in range(self.k):  
    self.centroids[i] = data[i]
```

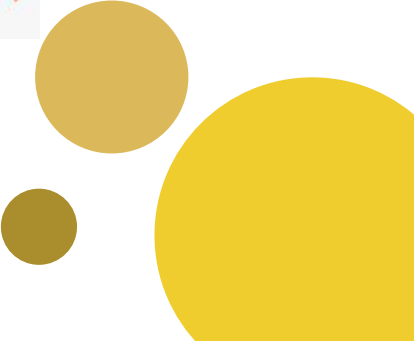


$O(k)$

```
for featureset in data:  
    distances = [np.linalg.norm(featureset-self.centroids[centroid])  
                 for centroid in self.centroids]  
    classification = distances.index(min(distances))  
    self.classifications[classification].append(featureset)
```




$O(N * k)$

- Space Complexity =  $\max(O(k)+O(n*k)) \approx O(n)$
  - For number of features in each data point =  $d$
  - Space Complexity can be written as  $O(n*d)$
- 



# K-Means Advantages

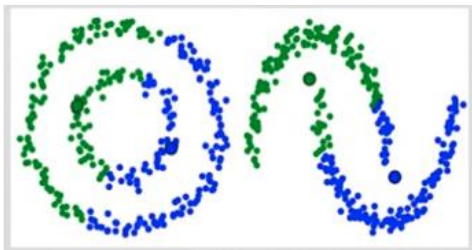
- It's pretty fast as all it does is compute the distance between points, It thus has a linear complexity  $O(tkn)$ , where  $n$  is number of points,  $k$  is number of clusters and  $t$  is number of iterations.
  - Relatively simple to implement.
  - Scales to large data sets.
  - Guarantees convergence.
  - Easily adapts to new examples.
  - Generalizes to clusters of different shapes and sizes, such as elliptical clusters.
- 



# K-Means

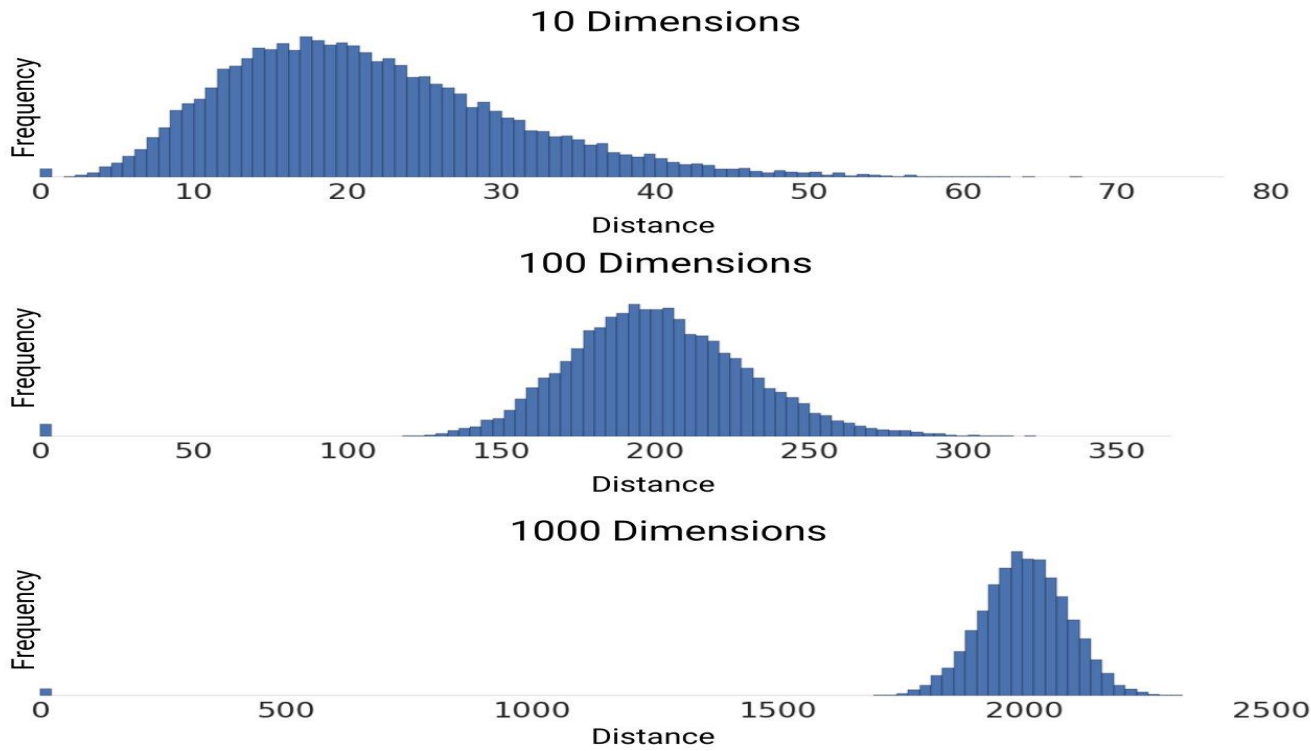
## Disadvantages

- Choosing K manually.
- Being dependent on initial values as it affects the final clusters.
- Sensitive to clustering outliers.
- Scaling with number of dimensions , As the number of dimensions increases, a distance-based similarity measure converges to a constant value between any given examples.
- K-means can't handle non-convex sets.



# K-Means

## Curse of Dimensionality





# K-Means Applications

Insurance Fraud Detection

Search Engines

Customer Segmentation

Image Segmentation

Diagnostic Systems

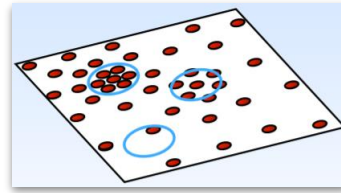
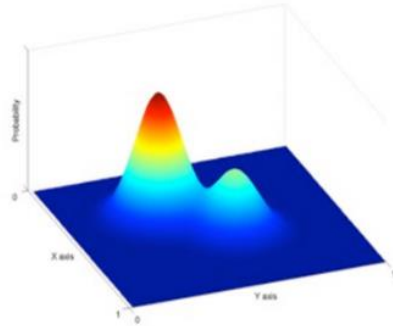




**Mean-Shift**

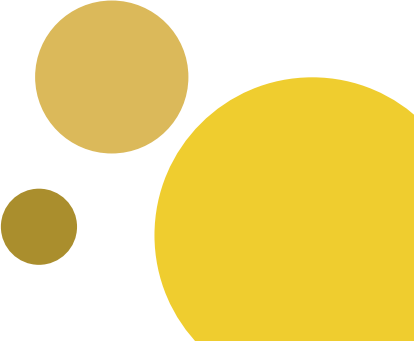
# Mean-Shift Definition

Mean-Shift is hierarchical density based clustering algorithm, it assigns the data points to the clusters iteratively by shifting points towards the mode (mode is the highest density of data points in the region, in the context of the Mean-Shift). As such, it is also known as the **Mode-seeking algorithm**.

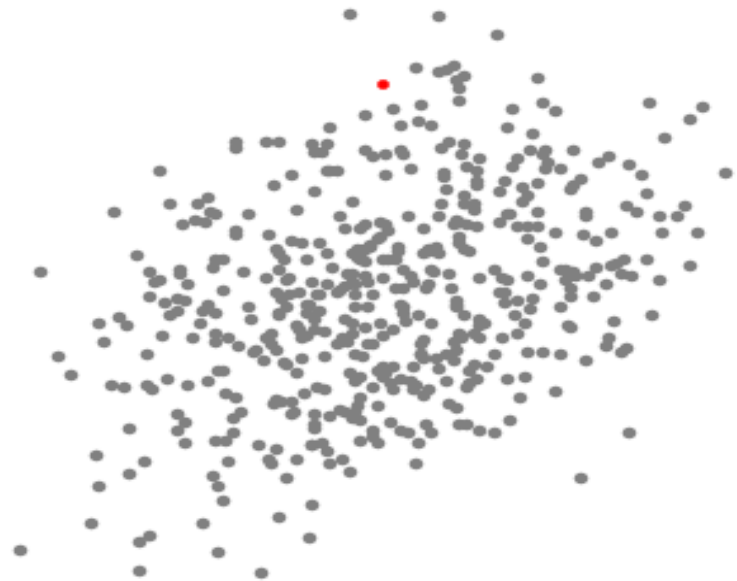
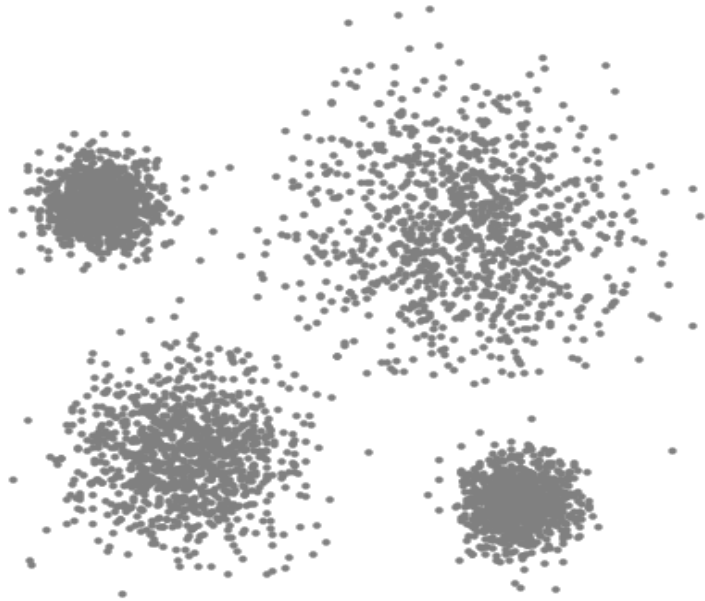




# Mean-Shift Steps

- **Step-1:** Start with the data points assigned to a cluster of their own.
  - **Step-2:** Calculate the mean of all the points lying inside this window.
  - **Step-3:** Shift the window, such that it is lying on the location of the mean.
  - **Step-4:** Iterate and move to the higher density region.
  - **Step-5:** Stop once the centroids reach at position from where it cannot move further.
- 

# Mean-Shift Animation



# Mean-Shift

## Code

```
def fit(self, data):
```

```
    # centroids : contains the updated centroids
    # set all input dataset points as a centroid for a cluster
    self.centroids = {}
    for i in range(len(data)):
        self.centroids[i] = data[i]

    while True:
        new_centroids = []
        for i in self.centroids:
            in_bandwidth = []
            centroid = self.centroids[i]

            # collect all points from the original data that are
            # within the radius of the current centroid in "in_bandwidth"
            for featureset in data:
                if np.linalg.norm(featureset-centroid) < self.radius:
                    in_bandwidth.append(featureset)

            # average the in_bandwidth points to get the new centroid and
            # save it in the "new_centroids" list
            new_centroid = np.average(in_bandwidth,axis=0)
            new_centroids.append(tuple(new_centroid))

        # save centroids before updates to terminate when no change occur
        prev_centroids = dict(self.centroids)
```

```
        # remove any coincide centroids in the new generated centroids and
        # update the centroid list
        uniques = sorted(list(set(new_centroids)))
        self.centroids = {}
        for i in range(len(uniques)):
            self.centroids[i] = np.array(uniques[i])
```

```
        # terminate when previous centroid list equals the new centroid list
        optimized = True
        for i in self.centroids:
            if not np.array_equal(self.centroids[i], prev_centroids[i]):
                optimized = False
            if not optimized:
                break
```

```
        if optimized:
            break
```

```
        self.classifications = {}
        in_bandwidth = {}
        for i in range(len(self.centroids)):
            self.classifications[i] = []
            in_bandwidth[i] = []
```

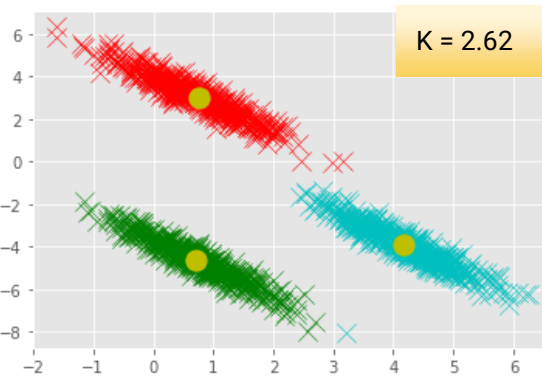
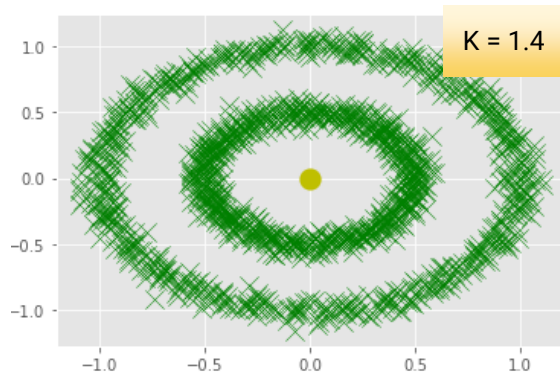
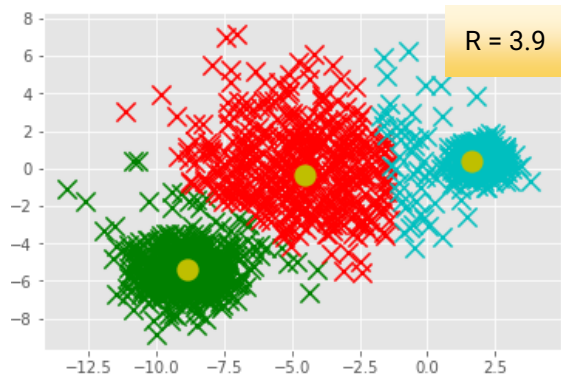
```
        for featureset in data:
            distances = [np.linalg.norm(featureset-self.centroids[centroid]) for centroid in self.centroids]
            classification = distances.index(min(distances))
            self.classifications[classification].append(featureset)
```

```
        print(in_bandwidth)
        return self.classifications,self.centroids
```

**Time :  $O(N^2)$**   
**Space :  $O(N)$**

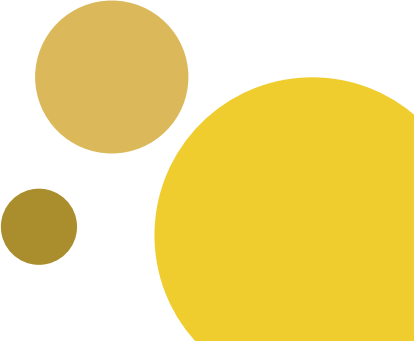


# Mean-Shift Results





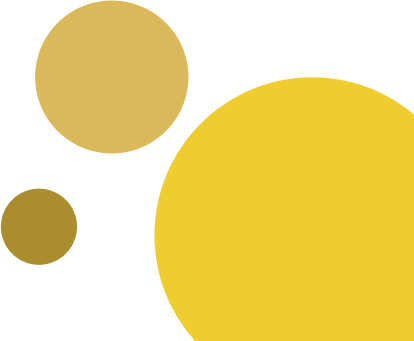
# Mean-Shift Advantages

- No need to provide number of clusters ( $k$ ) unlike k-means.
  - Automatically finds basins of attraction (maximum points)
  - One parameter choice (window size)
  - Robust to outliers.
  - Robust to noise within the data.
- 



# Mean-Shift

## Disadvantages

- Output depends on window size and window size (bandwidth) selection is not trivial.
  - Computationally expensive , as it has complexity of  $O(n^2)$ .
  - Does not scale well to large data sets and large dimensions of feature space.
- 

# Mean-Shift Applications

Clustering

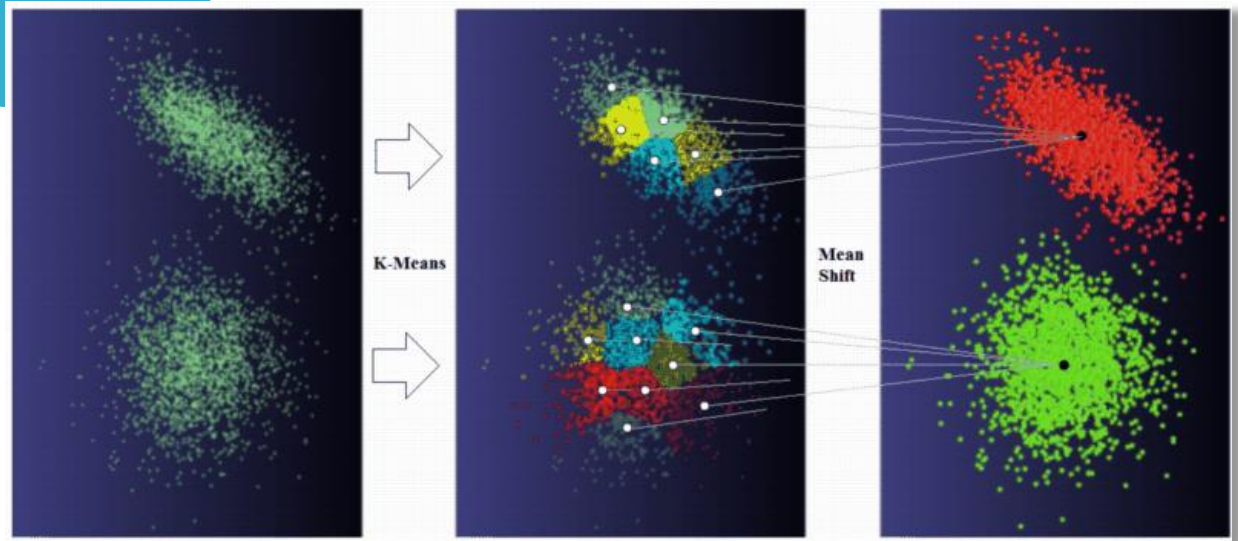
Object Contour Detection

Image Segmentation

Object Tracking



# Merging Mean-Shift with K-Means

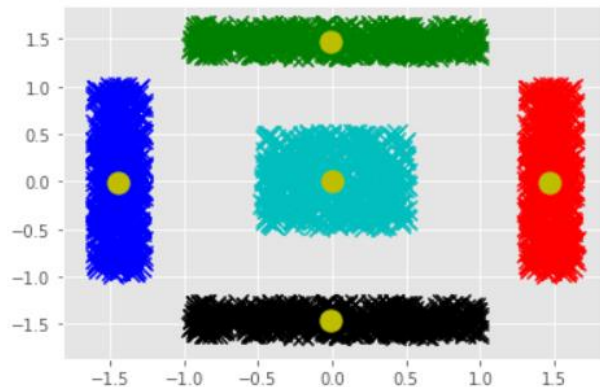


# K-Means

## Results ( K = 5 & K = 10 )

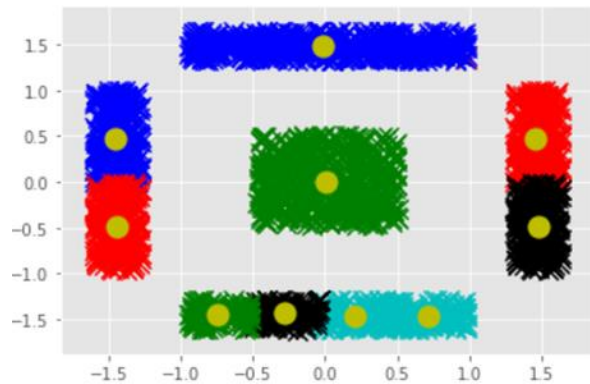
```
1 # perform K-means on dataset
2 tic = time.perf_counter()
3 kmeans = K_Means(k = 5)
4 classifications, centroids = kmeans.fit(Z)
5 toc = time.perf_counter()
6 print(f"K-means took {toc - tic:0.4f} seconds")
```

K-means took 0.9140 seconds



```
1 # perform K-means on dataset
2 tic = time.perf_counter()
3 kmeans = K_Means(k = 10)
4 classifications, centroids = kmeans.fit(Z)
5 toc = time.perf_counter()
6 print(f"K-means took {toc - tic:0.4f} seconds")
```

K-means took 2.3774 seconds



# K-Means & Mean-Shift

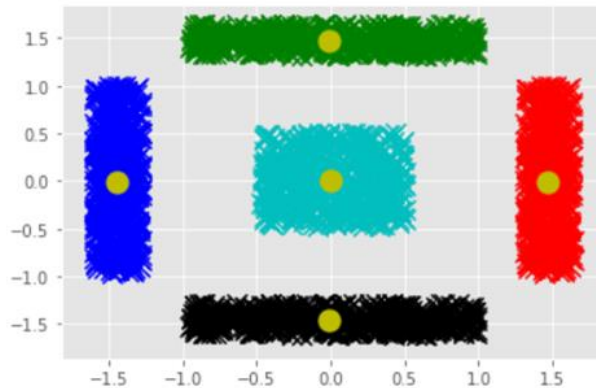
## Results ( K = 5 )

```
1 # perform K-means on dataset
2 tic = time.perf_counter()
3 kmeans = K_Means(k = 5)
4 classifications,centroids = kmeans.fit(Z)
5 toc = time.perf_counter()
6 print(f"K-means toke {toc - tic:0.4f} seconds")
7
8 # perform Mean shift on K-means resultant data
9 tic = time.perf_counter()
10 meanShift = Mean_Shift()
11 track = meanShift.fit(list(centroids.values()))
12 toc = time.perf_counter()
13 print(f"Mean shift toke {toc - tic:0.4f} seconds")
14
15 # map dataset points to resultant Mean shift clusters
16 tic = time.perf_counter()
17 clusters = Map_points_to_clusters(classifications,centroids)
18 toc = time.perf_counter()
19 print(f"Mapping point toke {toc - tic:0.4f} seconds")
```

K-means toke 0.8490 seconds

Mean shift toke 0.0010 seconds

Mapping point toke 0.0001 seconds



# K-Means & Mean-Shift

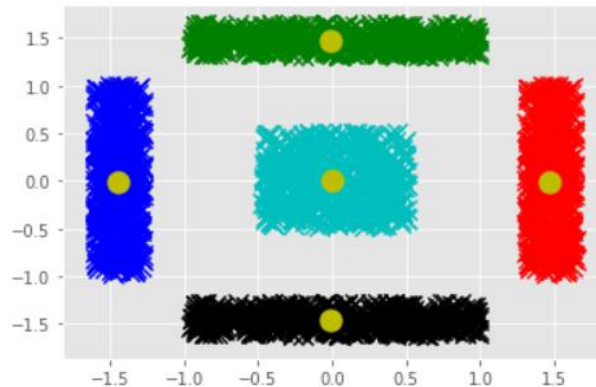
## Results ( K = 10 )

```
1 # perform K-means on dataset
2 tic = time.perf_counter()
3 kmeans = K_Means(k = 10)
4 classifications,centroids = kmeans.fit(Z)
5 toc = time.perf_counter()
6 print(f"K-means toke {toc - tic:0.4f} seconds")
7
8 # perform Mean shift on K-means resultant data
9 tic = time.perf_counter()
10 meanShift = Mean_Shift()
11 track = meanShift.fit(list(centroids.values()))
12 toc = time.perf_counter()
13 print(f"Mean shift toke {toc - tic:0.4f} seconds")
14
15 # map dataset points to resultant Mean shift clusters
16 tic = time.perf_counter()
17 clusters = Map_points_to_clusters(classifications,centroids)
18 toc = time.perf_counter()
19 print(f"Mapping point toke {toc - tic:0.4f} seconds")
```

K-means toke 2.0647 seconds

Mean shift toke 0.0030 seconds

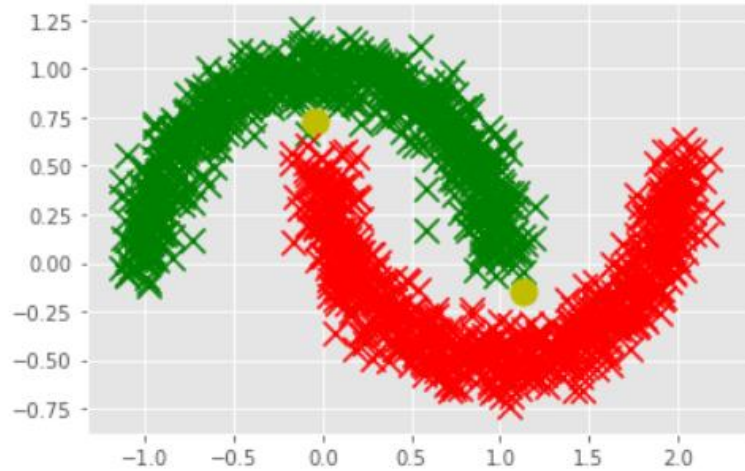
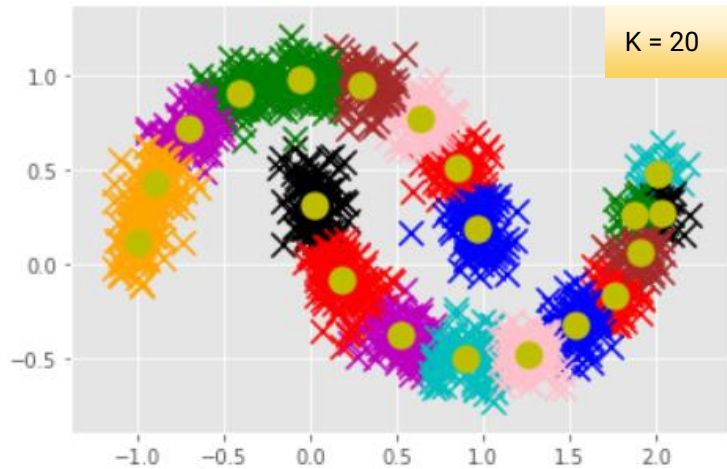
Mapping point toke 0.0001 seconds





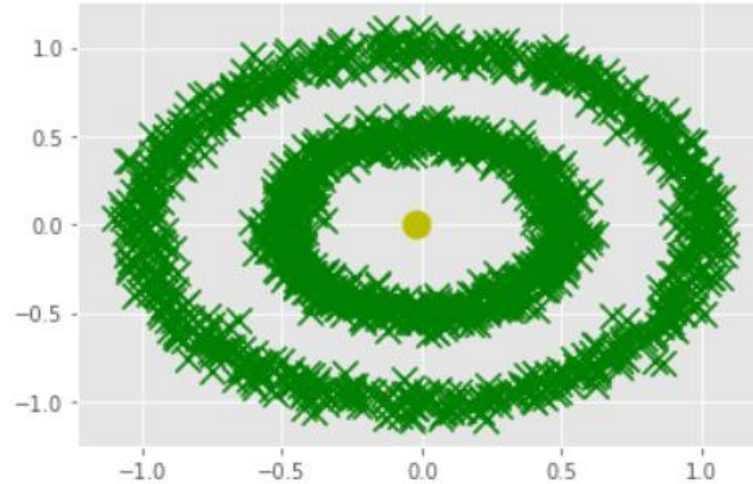
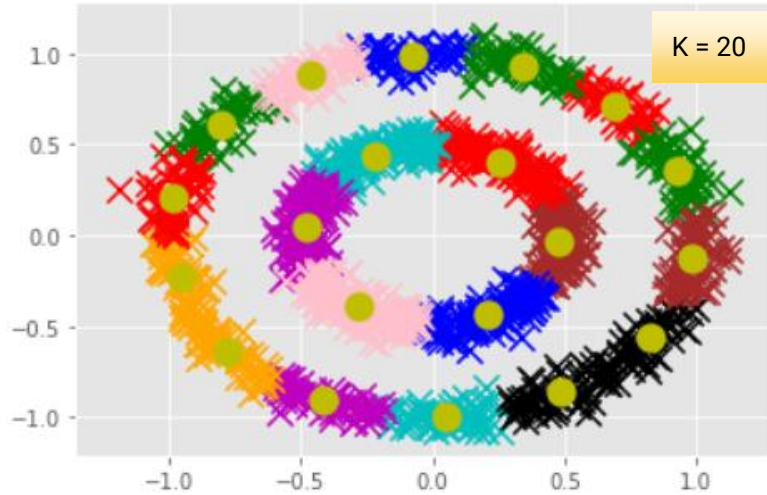
# K-Means & Mean-Shift

## Non-Convex Sets



# K-Means & Mean-Shift

## Concentric Sets





**DBSCAN**

# DBSCAN

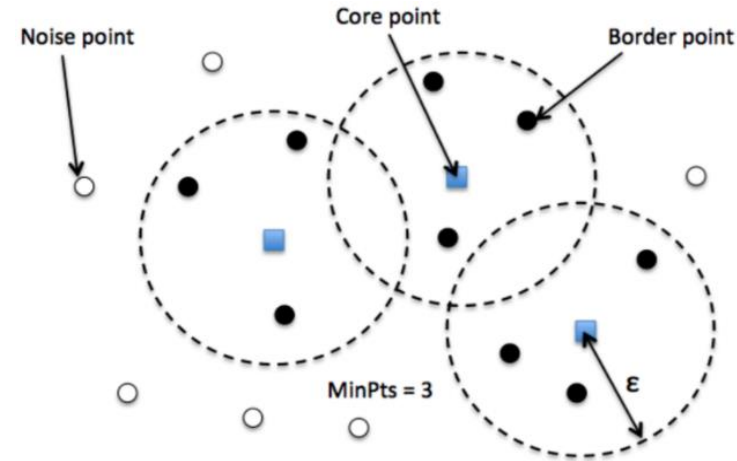
## Definition

**DBSCAN** ( Density-based spatial clustering of applications with noise ) is a density-based clustering.

**Two parameters:**

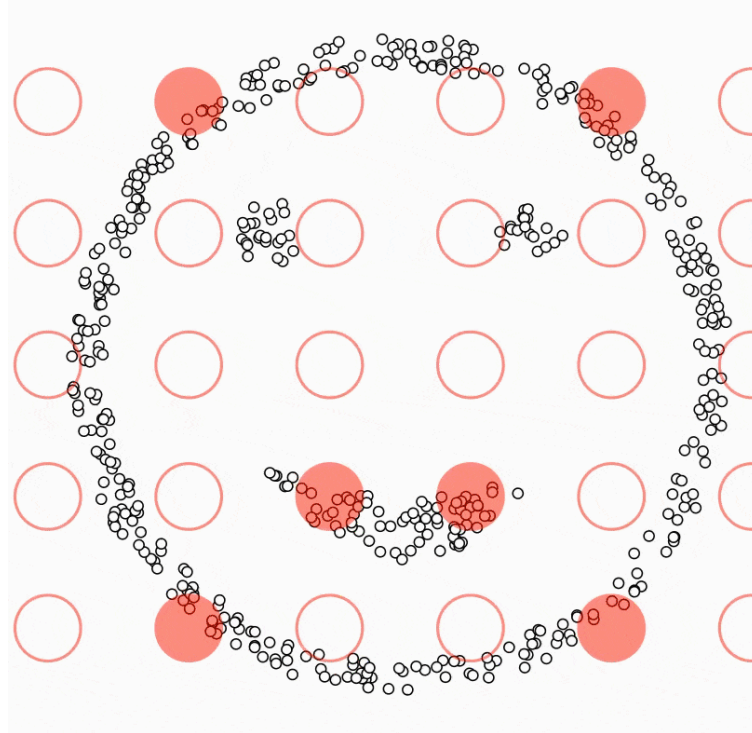
**Eps ( $\epsilon$ ):** Maximum radius of the neighborhood

**MinPts:** Minimum number of points in the Eps-neighborhood of a point



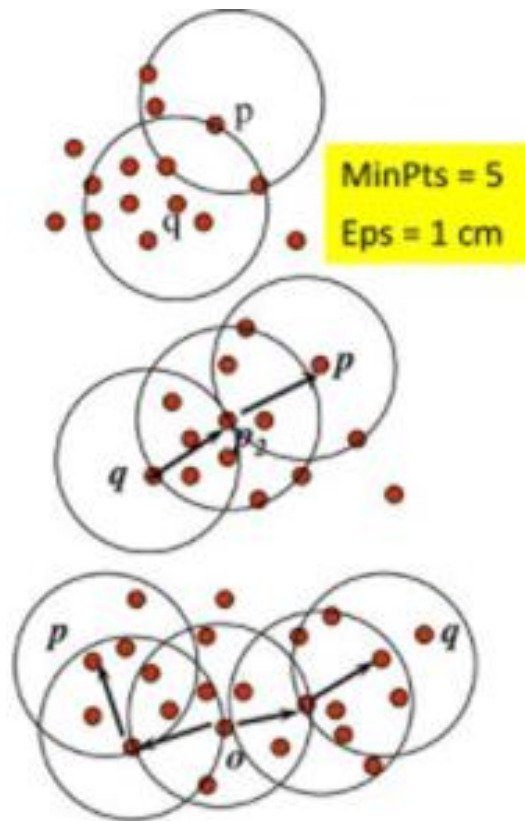
# DBSCAN

## Animation



# Density-Reachable and Density-Connected

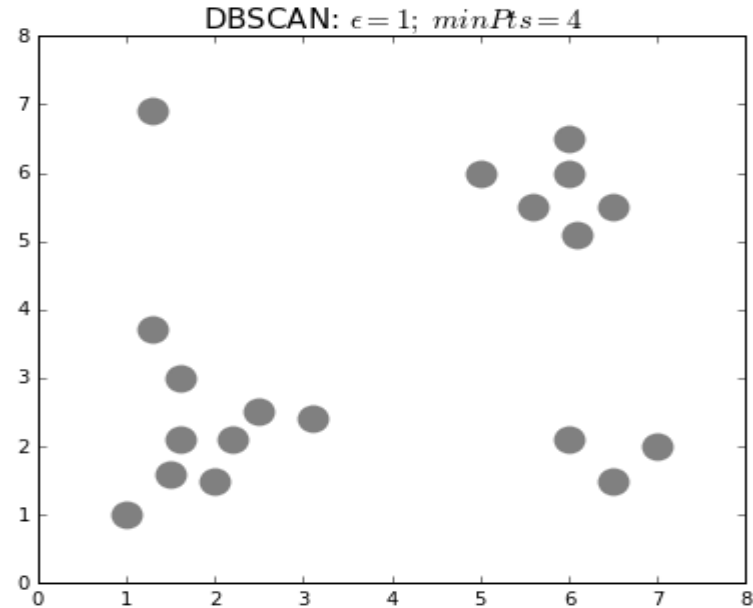
- Let  $q$  be a core point, then every point in its  $Eps$  neighborhood is said to be **directly density-reachable** from  $q$ .
- A point  $p$  is **density-reachable** from a point core point  $q$  if there is a chain of points.
- A point  $p$  is **density-connected** to a point  $q$  if there is a point  $o$  such that both,  $p$  and  $q$  are density-reachable from  $o$ .



# DBSCAN

## Steps

- **Step-1:** Find for a point in the dataset ,the neighbor points within eps and identify if it is outlier or the core points.
- **Step-2:** For each core point assigned it to a cluster.
- **Step-3:** Find all its density connected points and assign them to the same cluster as the core point.
- **Step-4:** Repeat until all all point assigned to cluster.





# DBSCAN

## Code

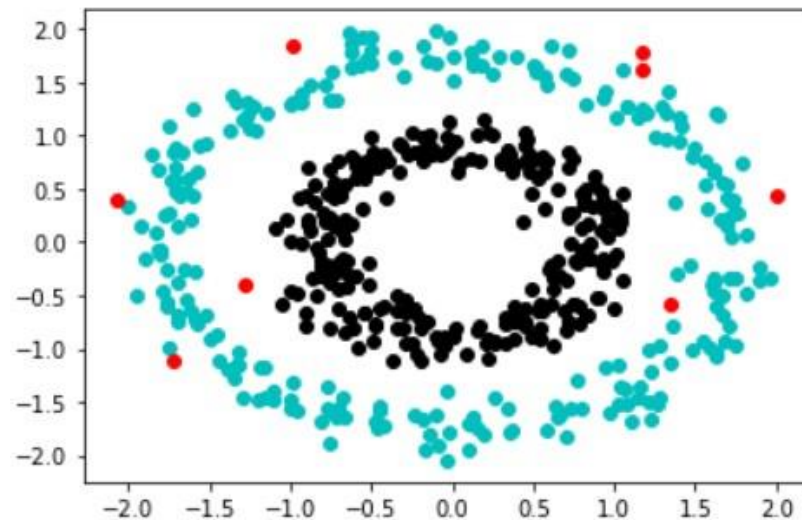
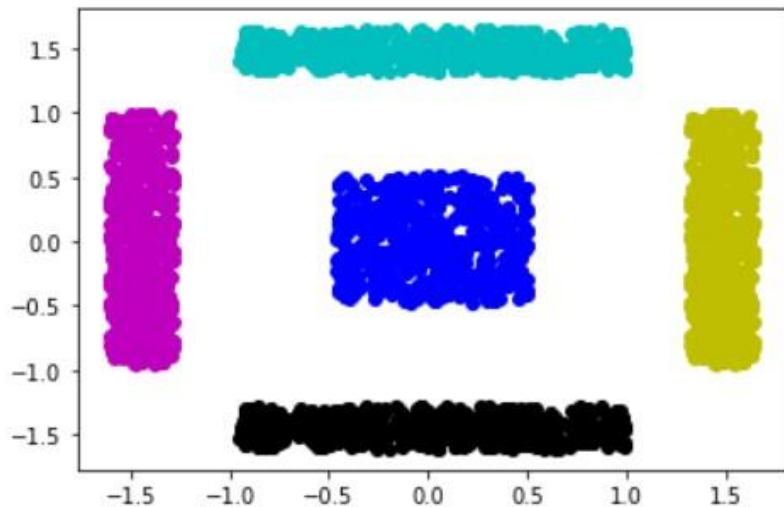
```
def fit(self,D, eps, MinPts):
    labels = [0]*len(D)
    C = 0
    # Iterate over each data point in the dataset
    for P in range(0, len(D)):
        # Check if this data point assigned to a cluster or not----
        # if assigned skip this point and choose another point----
        if not (labels[P] == 0):
            continue
        # go to RegionQuery function when the point
        # isnot assigned to any label yet (Label[P] is 0)
        NeighborPts = self.regionQuery(D, P, eps)
        # Then Check the number of neighbors
        if len(NeighborPts) < MinPts:
            # if the number less than Minpoints the it is Outlier!
            labels[P] = -1
        else:
            # if > Minpoints ----then our point is core point..
            # assign to the cluster a number and Grow the cluster
            # to include all points that achieve (eps, Minpoints)
            C += 1
            self.growCluster(D, labels, P, NeighborPts, C, eps, MinPts)
    return labels, len(set(labels))
```

```
def growCluster(self,D, labels, P, NeighborPts, C, eps, MinPts):
    # First assign the point to its new cluster..
    labels[P] = C
    i = 0
    # Iterate over all Our core point neighbor..
    while i < len(NeighborPts):
        Pn = NeighborPts[i]
        # Check If the neighbour point (in core point region)
        # Assigned before as outlier--- reassign it to our cluster
        if labels[Pn] == -1:
            labels[Pn] = C
            # Check too if this neighbour is still at intialization...
            # Assign it too to the new cluster
            elif labels[Pn] == 0:
                labels[Pn] = C
            # Then calculate the RegionQuery and get its neighbour List..
            PnNeighborPts = self.regionQuery(D, Pn, eps)
            # Check the neighbours list of each neighbour with the core eps..
            # If the number of its neighbour is > minpoints...
            # Then add this new list to the origin neighbour list of our core..
            # finally iterate for all points in the new list again..
            if len(PnNeighborPts) > MinPts:
                NeighborPts = NeighborPts + PnNeighborPts
            i += 1
    def regionQuery(self,D, P, eps):
        neighbors = []
        for Pn in range(0, len(D)):
            if np.linalg.norm(D[P] - D[Pn]) <= eps:
                neighbors.append(Pn)
        return neighbors
```



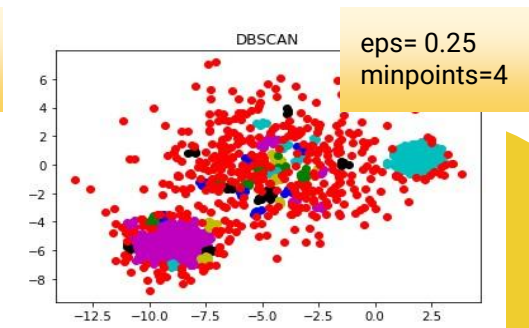
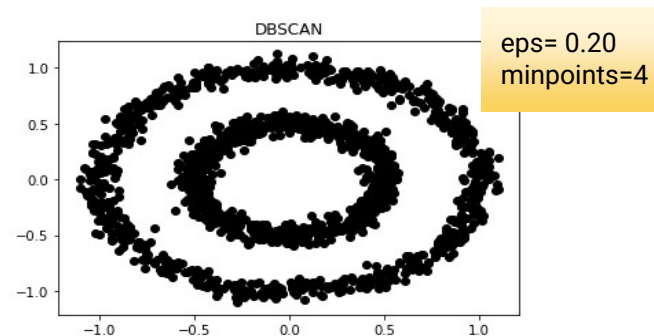
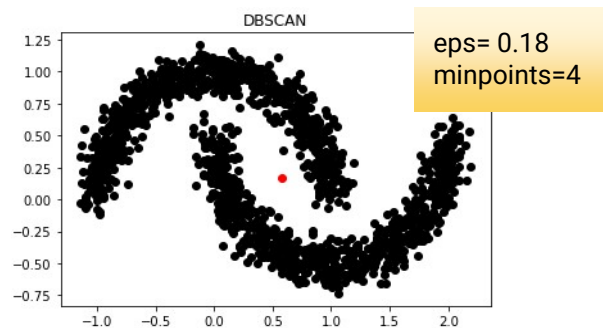
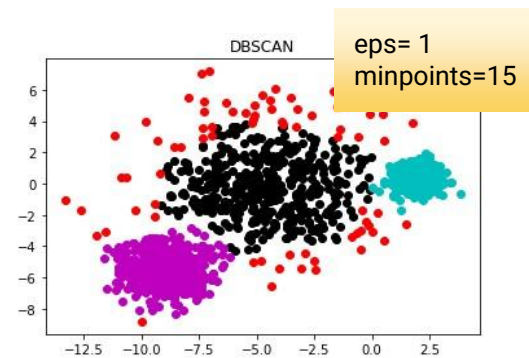
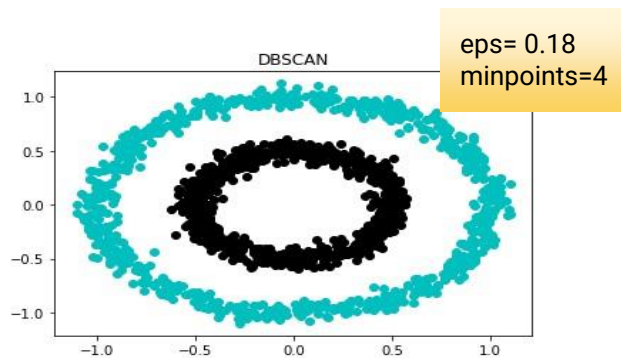
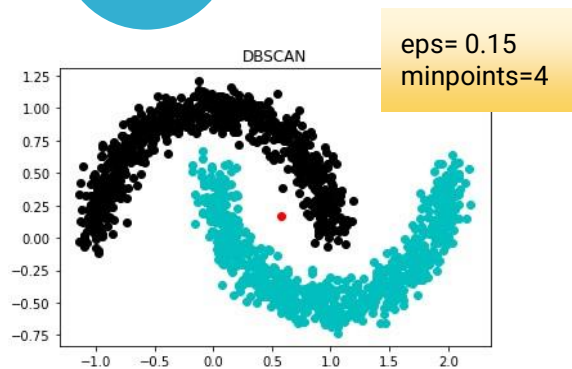
# DBSCAN

## Code Output



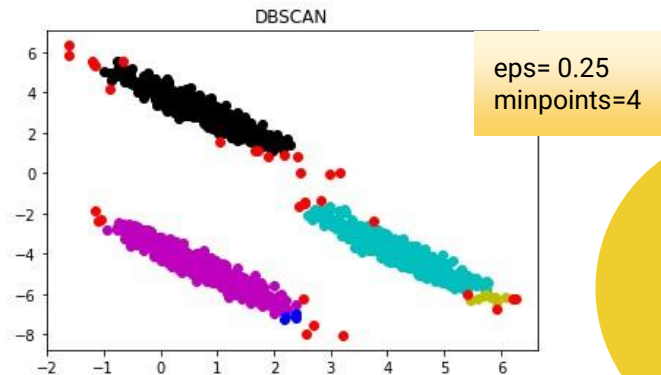
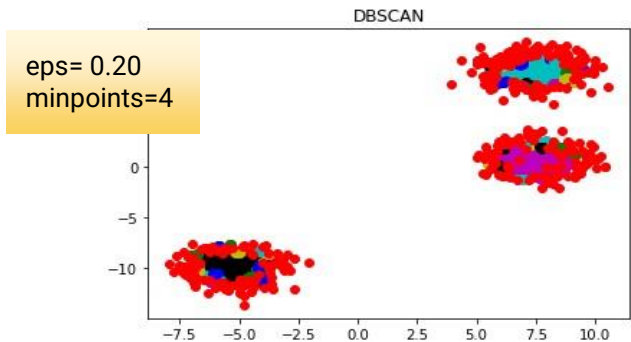
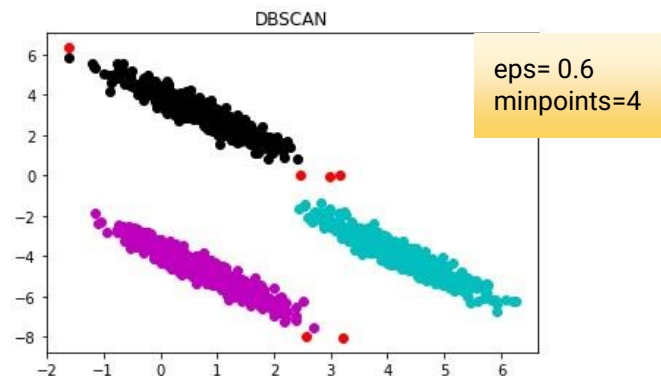
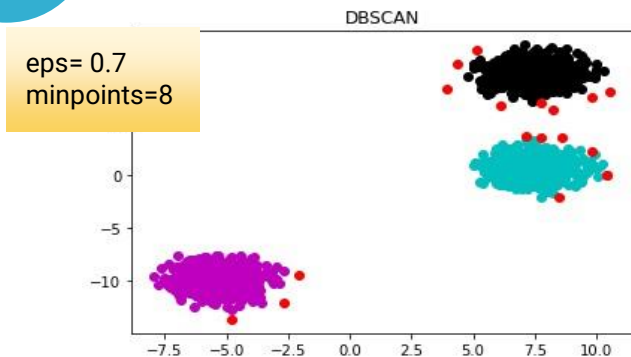
# DBSCAN

## Epsilon & MinPoints Effect



# DBSCAN

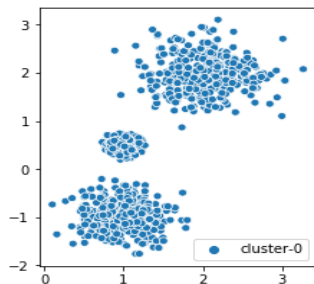
## Epsilon & MinPoints Effect



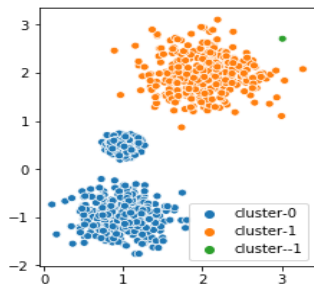
# DBSCAN

## Epsilon Effect

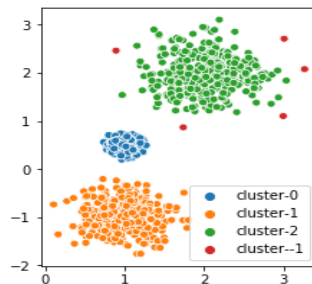
eps = 1.0



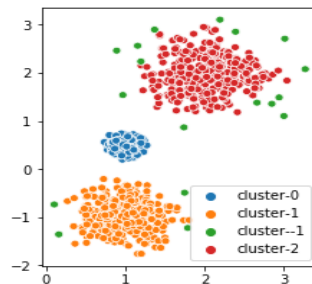
eps = 0.5



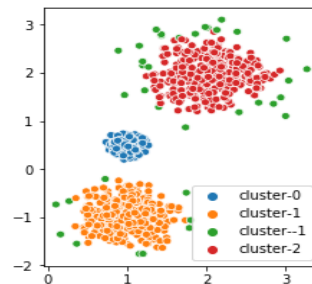
eps = 0.33



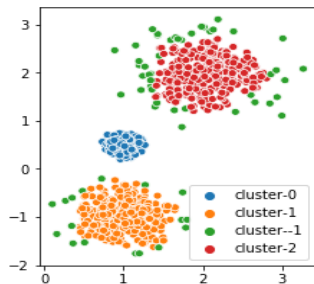
eps = 0.25



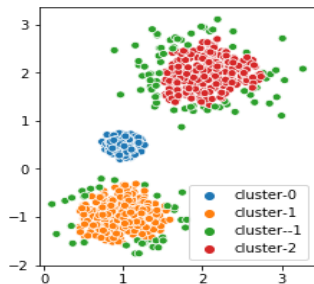
eps = 0.2



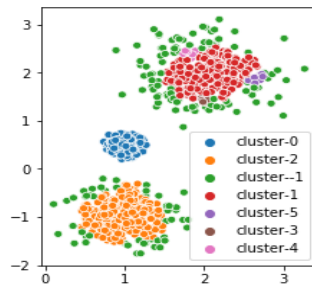
eps = 0.17



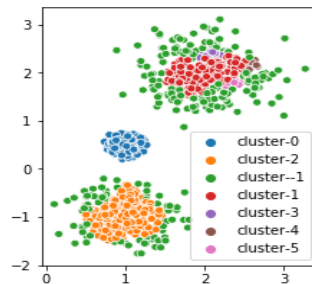
eps = 0.14



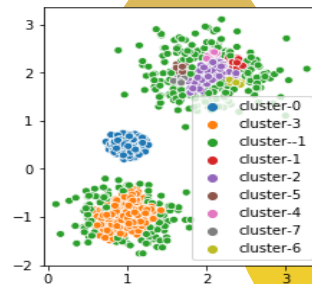
eps = 0.12



eps = 0.11



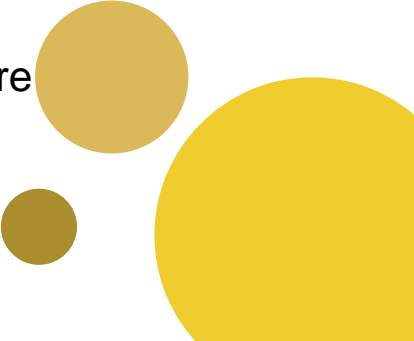
eps = 0.1





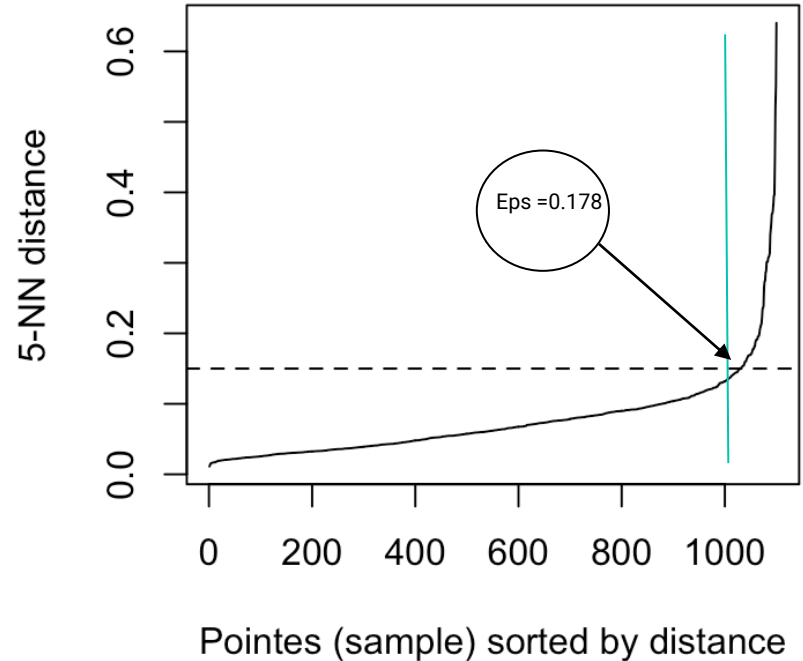
# “MinPts” Selection

**Here are a few rules of thumb for selecting the MinPts value:**

- The larger the data set, the larger the value of MinPts should be.
  - If the data set is noisier, choose a larger value of MinPts.
  - Generally, MinPts should be greater than or equal to the dimensionality of the data set.
  - For 2-dimensional data, use DBSCAN's default value of  $\text{MinPts} = 4$ .
  - If your data has more than 2 dimensions, choose  $\text{MinPts} = 2 \cdot \text{dim}$ , where  $\text{dim}$  = the dimensions of your data set.
- 

# Epsilon ( $\epsilon$ ) Selection

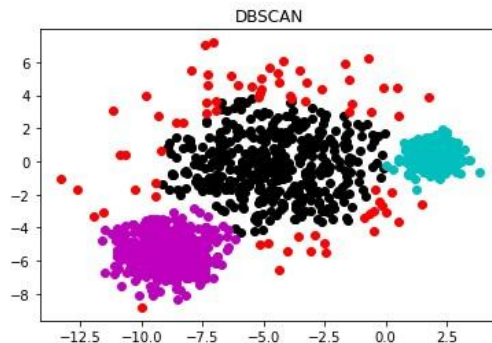
- KNN calculates the average distance between each point and its  $k$  nearest neighbors.
- The average  $k$ -distances are then plotted in ascending order on a  $k$ -distance graph.
- You'll find the optimal value for  $\epsilon$  at the point of maximum curvature (i.e. where the graph has the greatest slope).



# DBSCAN

## Advantages

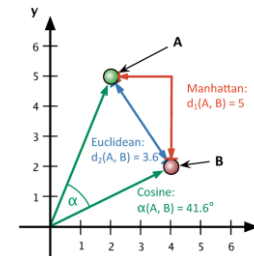
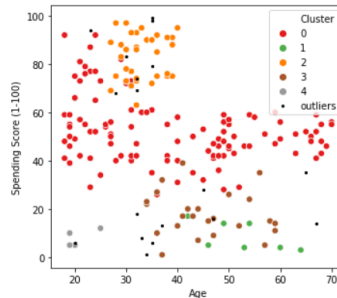
- Number of clusters is not required.
- It can find any shape cluster even if the cluster is surrounded by any other cluster.
- It can easily find outliers in data set.
- It is the second most used clustering method after K-means.



# DBSCAN

## Disadvantages

- The quality of the result depends on the distance measure used in the regionQuery function.
- It can be expensive when cost of computation of nearest neighbor is high.
- It can be slow in execution for higher dimension.
- Not Suitable if dataset is too sparse.
- Sensitive to eps and Minpts.





# DBSCAN

## Code Analysis

n	$f(n) = n^2$	cost
500	250000	0.25 us
1000	1000000	1 us

Assume: • Frequency = 1Mhz •  $\therefore$  instruction =  $1 \mu s$

Frequency Count Method

```
class CustomDBSCAN:
    def __init__(self):
        pass
    def fit(self, D, eps, MinPts):
        labels = [0]*len(D)
        C = 0
        for P in range(0, len(D)):
            if not (labels[P] == 0):
                continue
            NeighborPts = self.regionQuery(D, P, eps)
            if len(NeighborPts) < MinPts:
                labels[P] = -1 ## Outlier
            else:
                C += 1
                self.growCluster(D, labels, P, NeighborPts, C, eps, MinPts)
        return labels, len(set(labels))
```

Annotations for fit method:

- `len(D)`:  $n$
- `[0]*len(D)`:  $1$
- `range(0, len(D))`:  $n$
- `len(NeighborPts)`:  $n$
- `len(D)` (in regionQuery):  $n$
- `len(set(labels))`:  $1$
- `C += 1`:  $2n$

$$f(n) = 8n^2 + 8nd + 12n + 2 \cong f(n) = n^2 \rightarrow O(n^2)$$

Complexity Space =  $O(n)$

```
def growCluster(self, D, labels, P, NeighborPts, C, eps, MinPts):
    labels[P] = C
    i = 0
    while i < len(NeighborPts):
        Pn = NeighborPts[i]
        if labels[Pn] == -1:
            labels[Pn] = C
        elif labels[Pn] == 0:
            labels[Pn] = C
            PnNeighborPts = self.regionQuery(D, Pn, eps)
            if len(PnNeighborPts) > MinPts:
                NeighborPts = NeighborPts + PnNeighborPts
        i += 1
```

Annotations for growCluster method:

- `labels[P] = C`:  $n$  (it is 1 but inside for loop then  $n$ )
- `len(NeighborPts)`:  $n$
- `NeighborPts[i]`:  $n$
- `len(PnNeighborPts)`:  $n$
- `NeighborPts = NeighborPts + PnNeighborPts`:  $n \cdot 2d$

```
def regionQuery(self, D, P, eps):
    neighbors = []
    for Pn in range(0, len(D)):
        if np.linalg.norm(D[P] - D[Pn]) <= eps:
            neighbors.append(Pn)
    return neighbors
```

Annotations for regionQuery method:

- `len(D)`:  $n$
- `range(0, len(D))`:  $n$
- `np.linalg.norm(D[P] - D[Pn])`:  $n \cdot n$
- `neighbors.append(Pn)`:  $n \cdot n$



# DBSCAN Applications

Scientific Literature

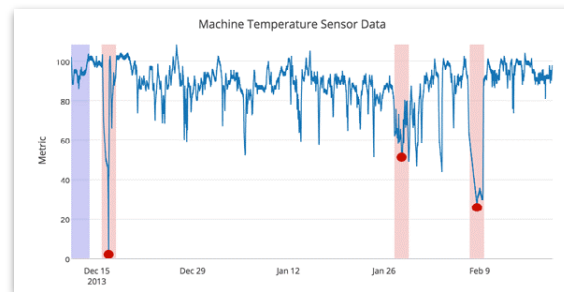
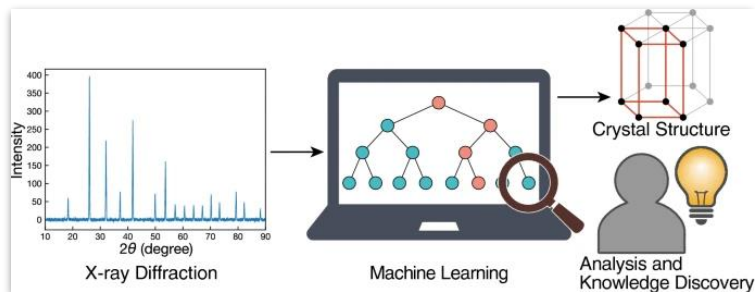
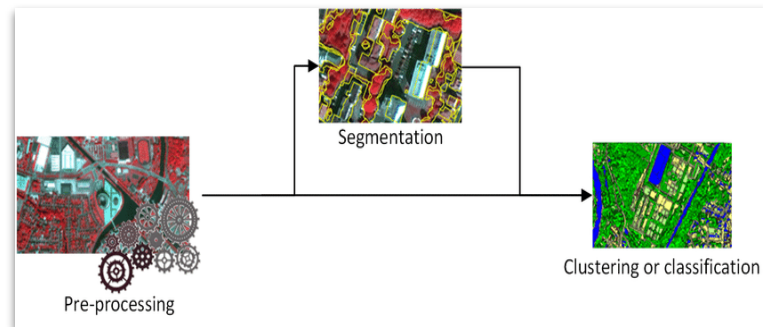
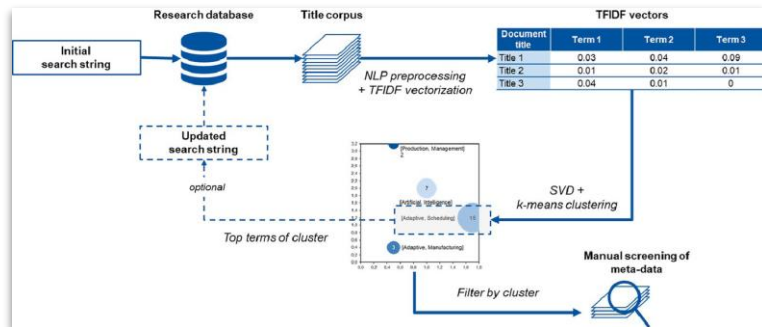
Image of Satellite

Crystallography of X-Ray

Anomaly Detection in Temperation Data



# DBSCAN Applications



# Conclusion

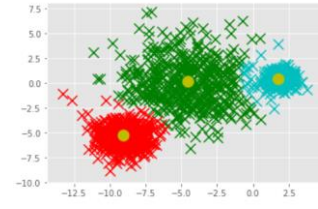
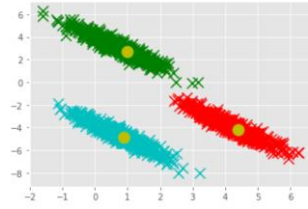
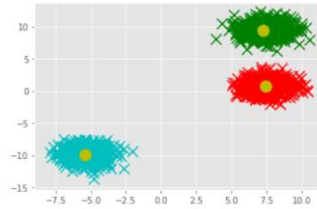
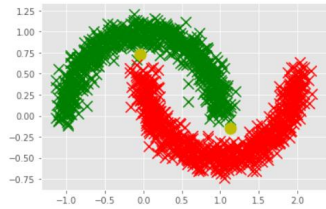
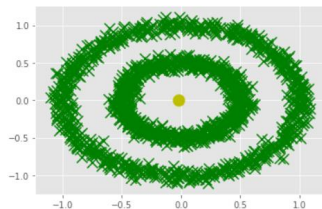
## K-Means/Mean-Shift

## DBSCAN

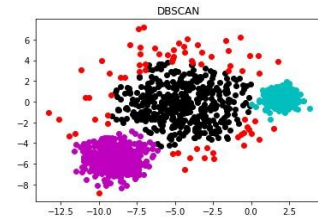
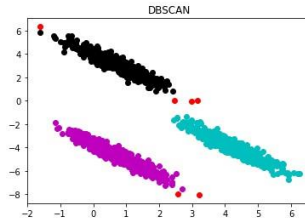
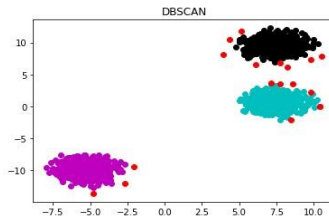
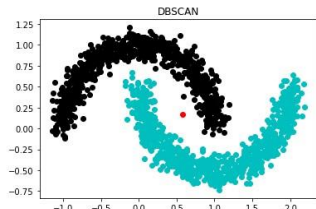
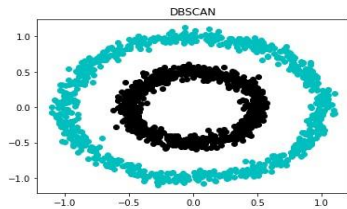
Time Complexity	$O(nl)$	$O(n^2)$
Clustering Type	Partitional Clustering	Density Based Clustering
Paradigm	Greedy Algorithm	Greedy Algorithm
Outlier Detection	Can not Handle Outliers	Can Handle Outliers

# Conclusion

## K-Means/Mean-Shift

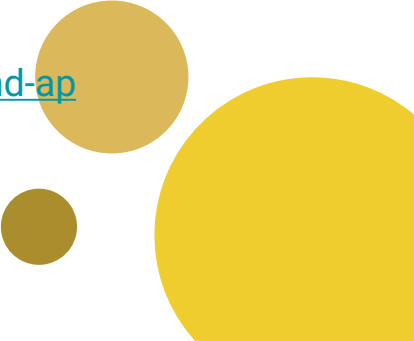


## DBSCAN





# References

- <http://jamesxli.blogspot.com/2012/03/on-mean-shift-and-k-means-clustering.html>
  - <https://www.geeksforgeeks.org/dbscan-full-form/>
  - <https://www.tutorialspoint.com>
  - <https://en.wikipedia.org>
  - <https://www.analyticsvidhya.com/blog/2020/09/how-dbscan-clustering-works/>
  - <https://developers.google.com/machine-learning/clustering/algorithm/advantages-disadvantages>
  - <https://towardsdatascience.com/how-to-use-dbscan-effectively-ed212c02e62>
  - <https://www.slideshare.net/MahbuburShimul/dbscan-algorithm>
  - <https://www.kaggle.com/datark1/customers-clustering-k-means-dbscan-and-ap>
- 



**Any Questions ?**





**THANKS!**