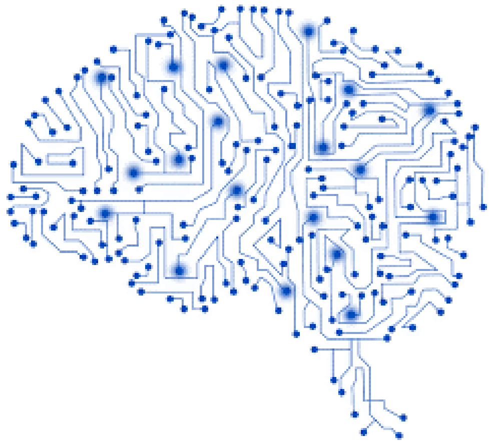
**University of Minho**
School of Engineering

Intelligent Systems Associate Laboratory



# Dados e Aprendizagem Automática

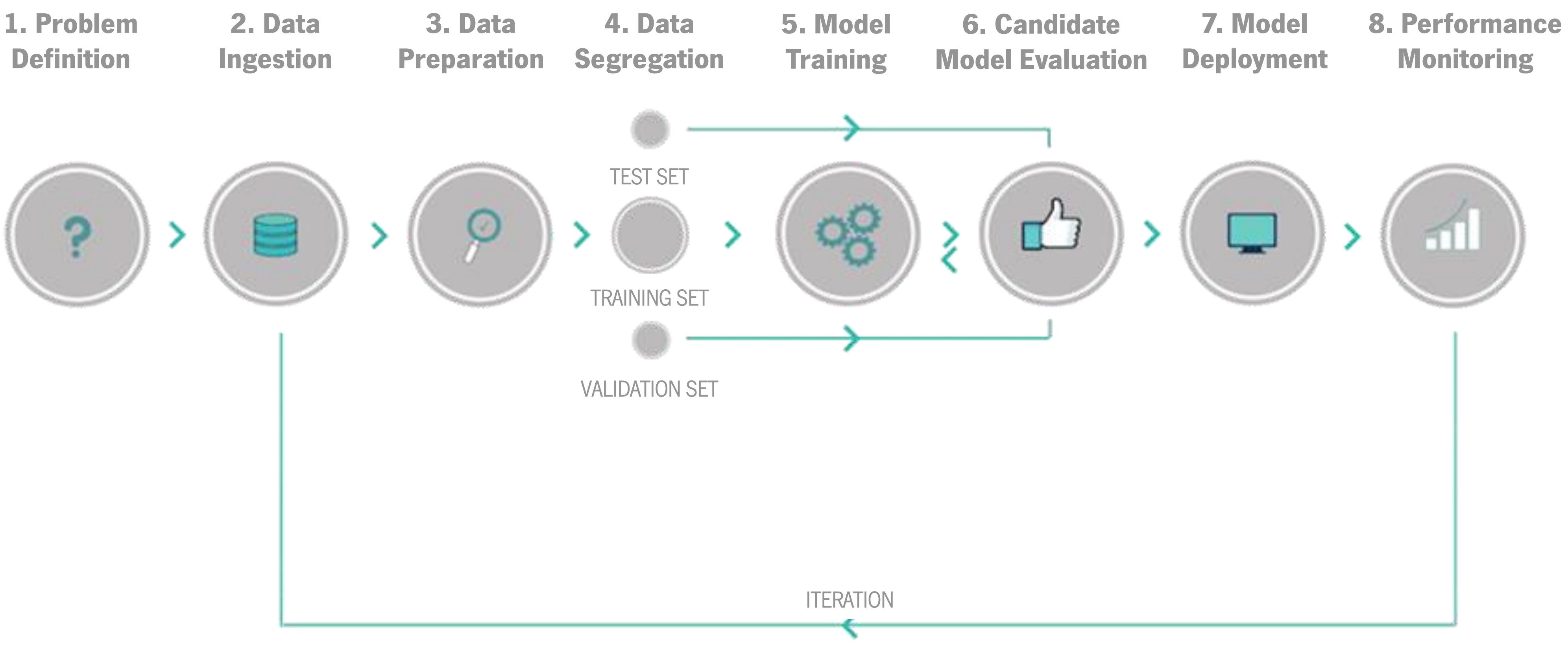
## Unsupervised Learning:
## K-means, K-medoids and DBSCAN

**DAA @ MEI-1º/MiEI-4º – 1º Semestre**

Bruno Fernandes, Dalila Alves, Filipa Ferraz, Victor Alves

*Part IX*

# Contents


1. Problem Definition
2. Data Ingestion
3. Data Preparation
4. Data Segregation
5. Model Training
6. Candidate Model Evaluation
7. Model Deployment
8. Performance Monitoring
TEST SET
TRAINING SET
VALIDATION SET
ITERATION

# Unsupervised Learning

# Unsupervised Learning



**Unsupervised learning** means that there is no outcome to be predicted, and the algorithm just tries to find patterns on the data. Using **clustering**, it tries to group (cluster) the data based on their similarity.

For this example, we create the data creating *isotropic Gaussian blobs* from `sklearn.datasets`.

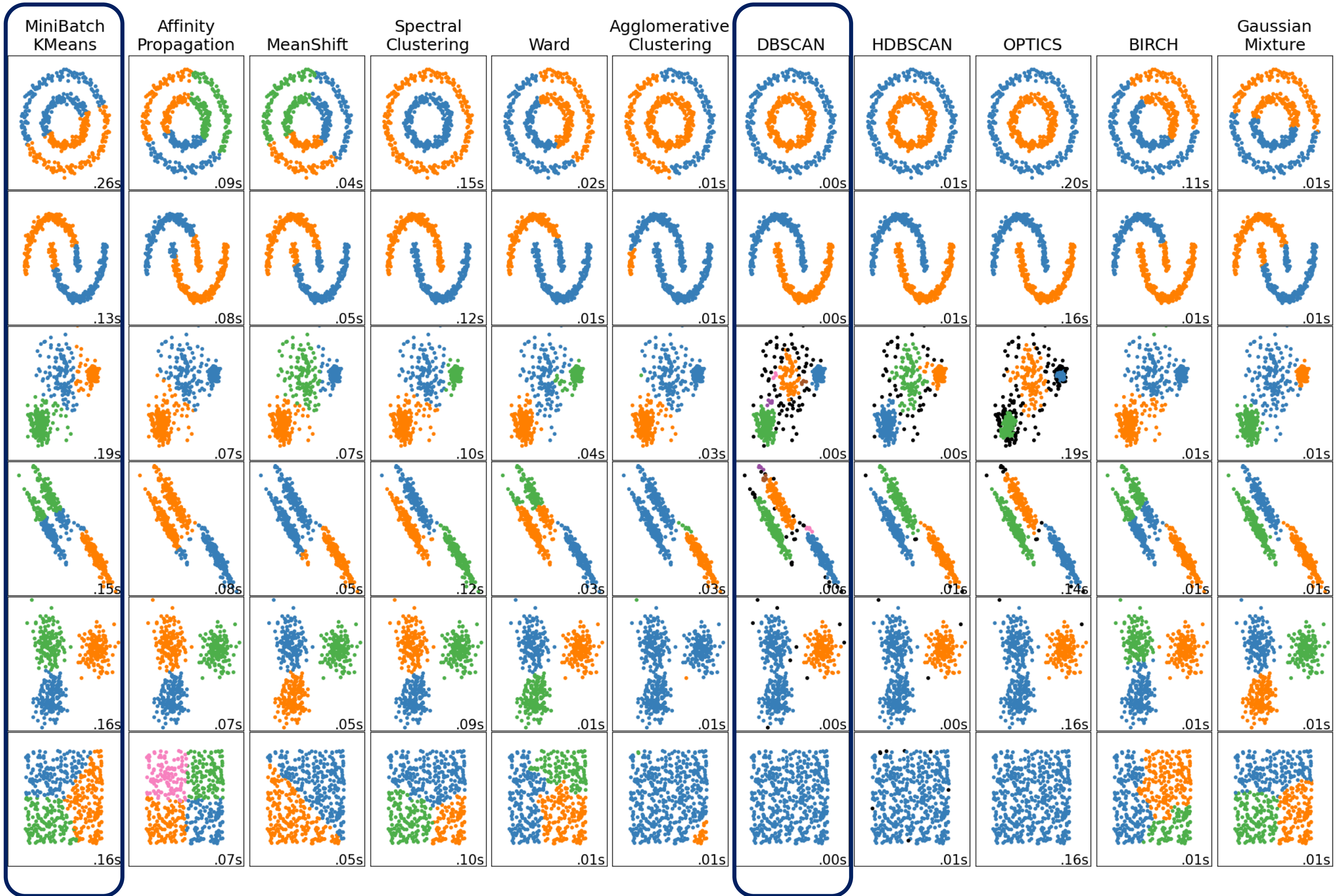
Note: all clustering algorithms require data preprocessing(e.g. dimensionality reduction) and standardization.

You may need to install `scikit-learn-extra`.

```
pip install https://github.com/scikit-learn-contrib/scikit-learn-extra/archive/master.zip
```

# Unsupervised Learning - Clustering

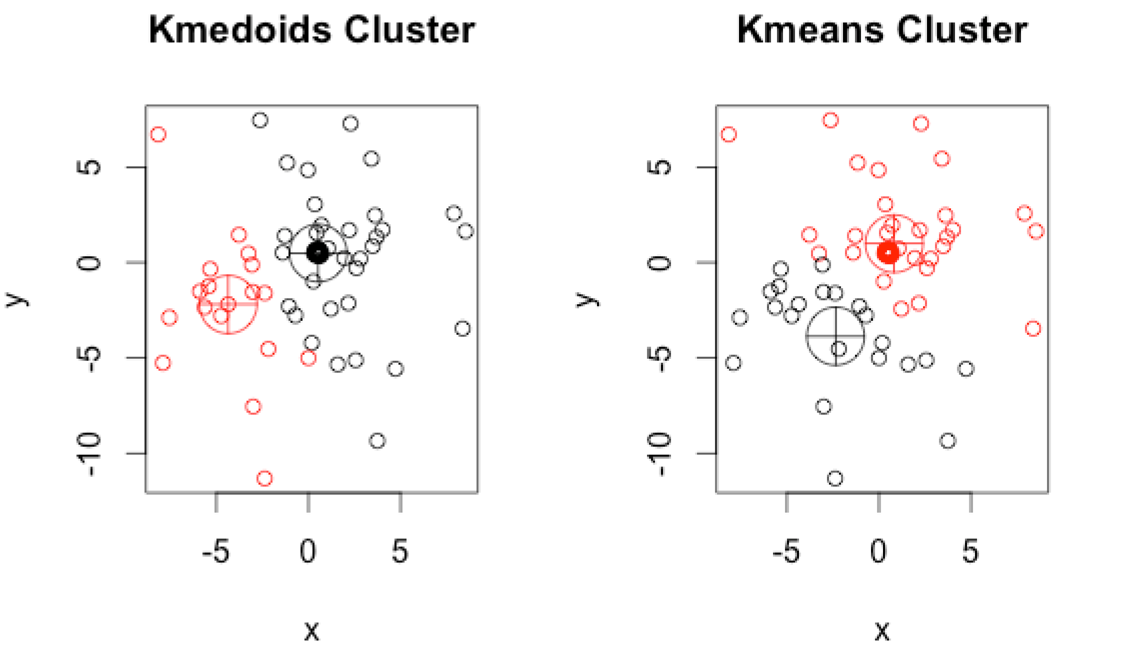




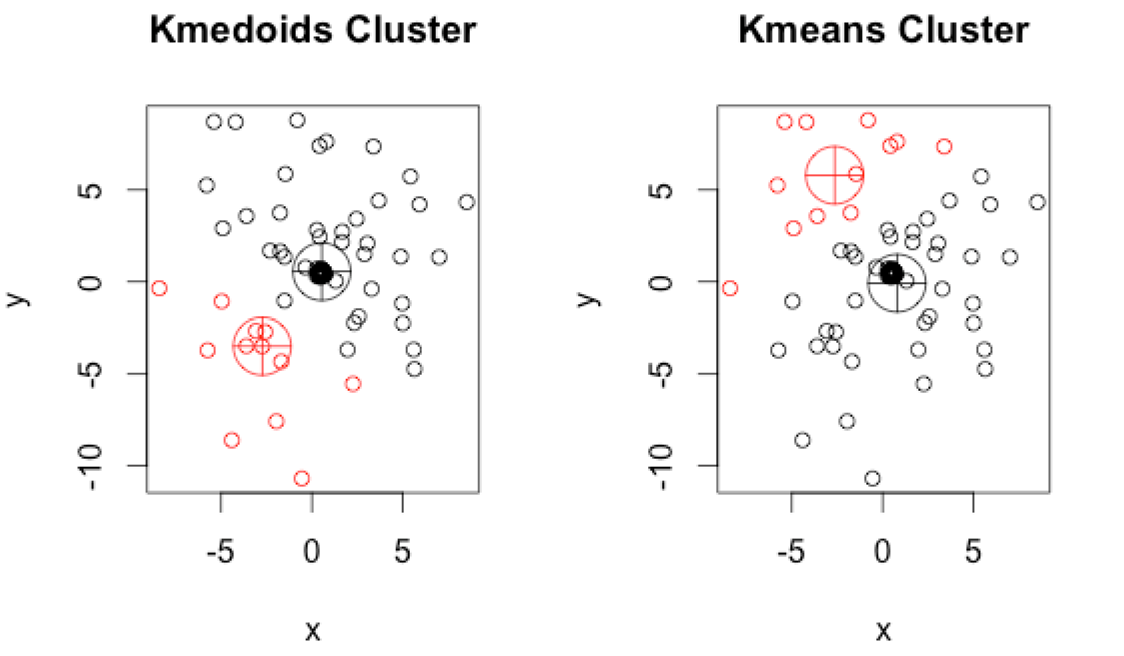
# Unsupervised Learning – K-Means and K-Medoids



1st iteration




2nd iteration




https://stats.stackexchange.com/questions/156210/an-example-where-the-output-of-the-k-medoid-algorithm-is-different-than-the-outp

# Unsupervised Learning – K-Means and K-Medoids



Both **K-Means** and **K-Medoids** algorithms follow the **partitioning method**. So they are:

- breaking the dataset up into *k* groups;
- trying to minimize the distance between points of the same cluster and a particular point which is the center of that cluster.

The **K-Means** algorithm chooses *centroids*, the geometric center of a cluster using the mean to all cluster points.

On the other hand, the **K-Medoids** algorithm chooses *medoids*, points as centers that belong to the dataset and computes dissimilarities between data points.

The most common implementation of the K-Medoids clustering algorithm is the **Partitioning Around Medoids (PAM)** algorithm. The PAM algorithm uses a *greedy search*, which may not find the global optimum solution.

# Unsupervised Learning – K-Means and K-Medoids



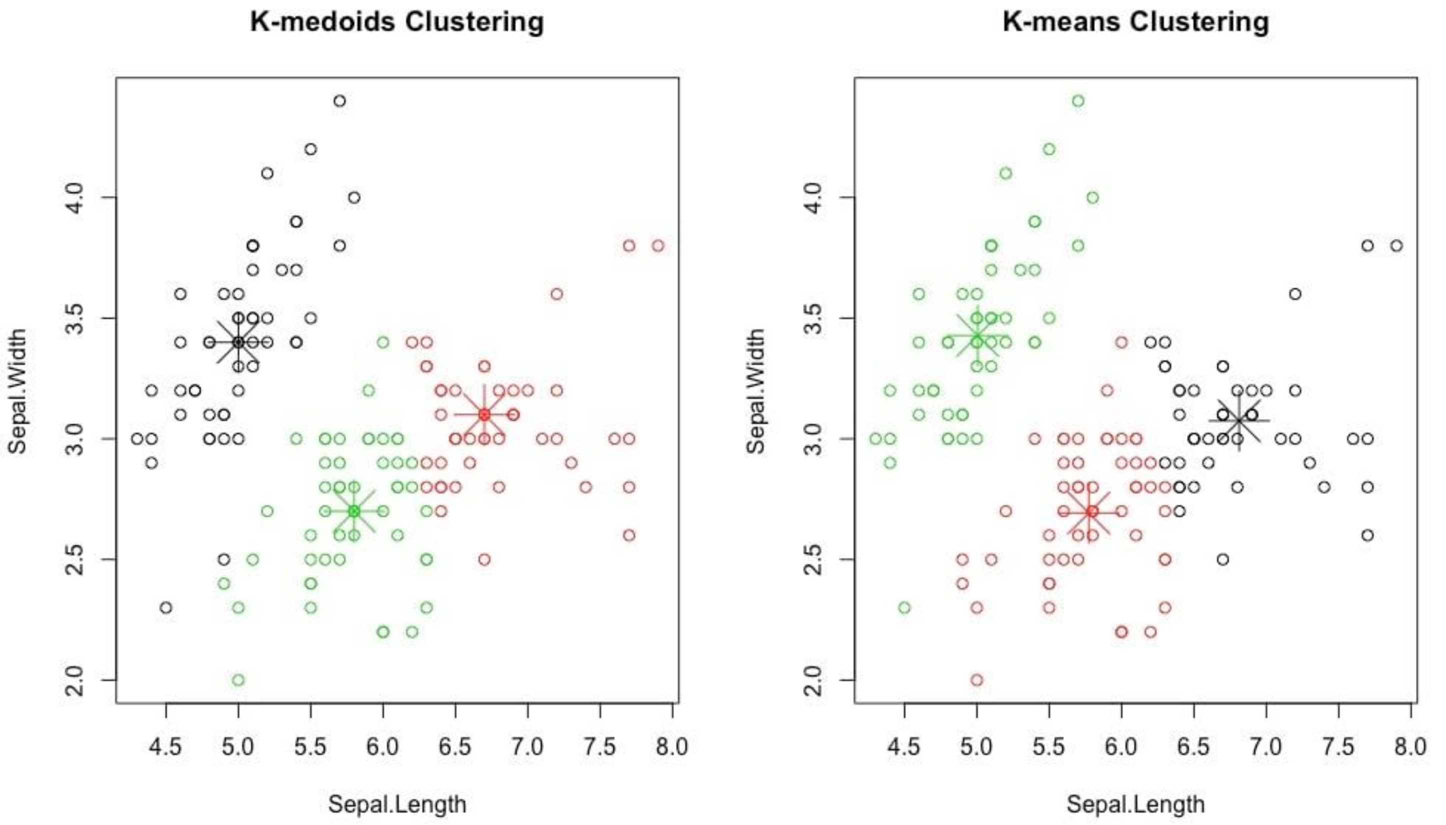
*Medoids* are more robust to outliers than *centroids*, but they need more computation for high dimensional data.

Strengths:
- simple and intuitive;
- scales to large datasets;
- as a result, we also have centroids that can be used as standard cluster representatives

Weaknesses:
- knowledge about the number of clusters is necessary and must be specified as a parameter;
- does not cope well with a very large number of features;
- separates only convex and homogeneous clusters well;
- can result in poor local solutions, so it needs to be run several times.

# Unsupervised Learning – K-Means and K-Medoids

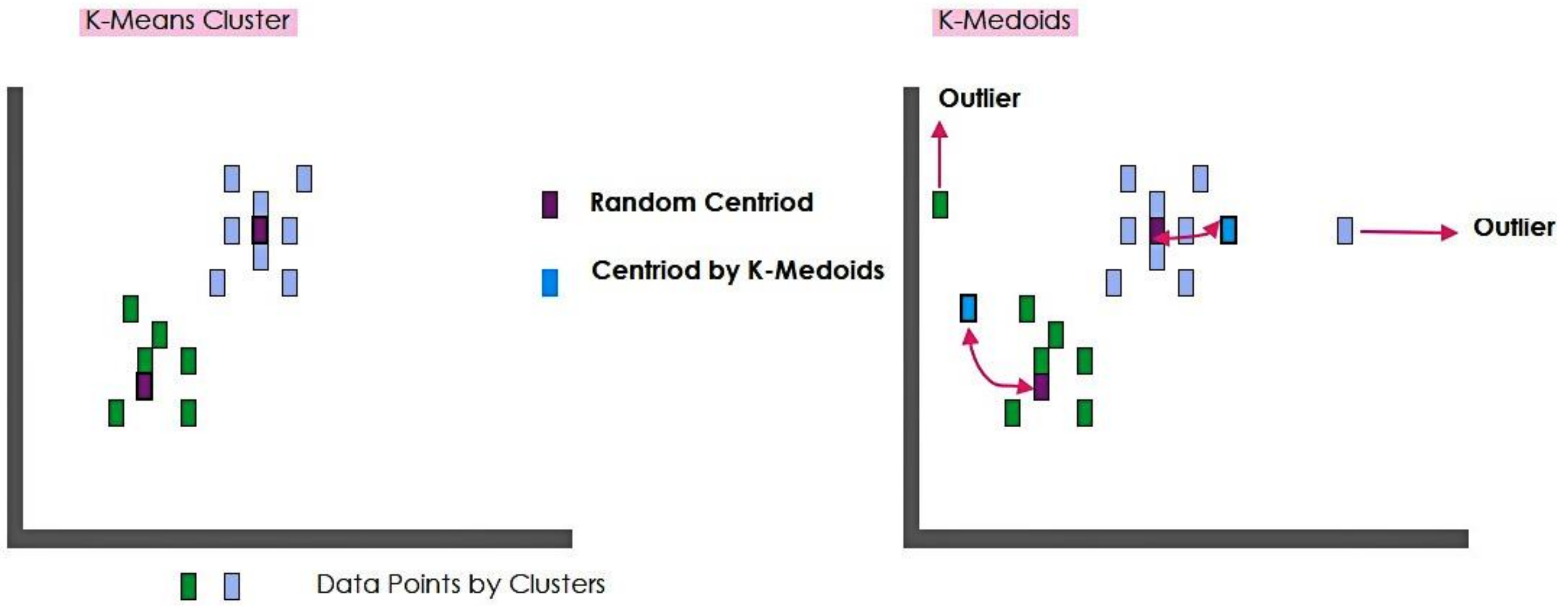





https://subscription.packtpub.com/book/big-data-and-business-intelligence/9781789956399/1/ch01lvl1sec08/introduction-to-k-medoids-clustering

# Unsupervised Learning – K-Means and K-Medoids





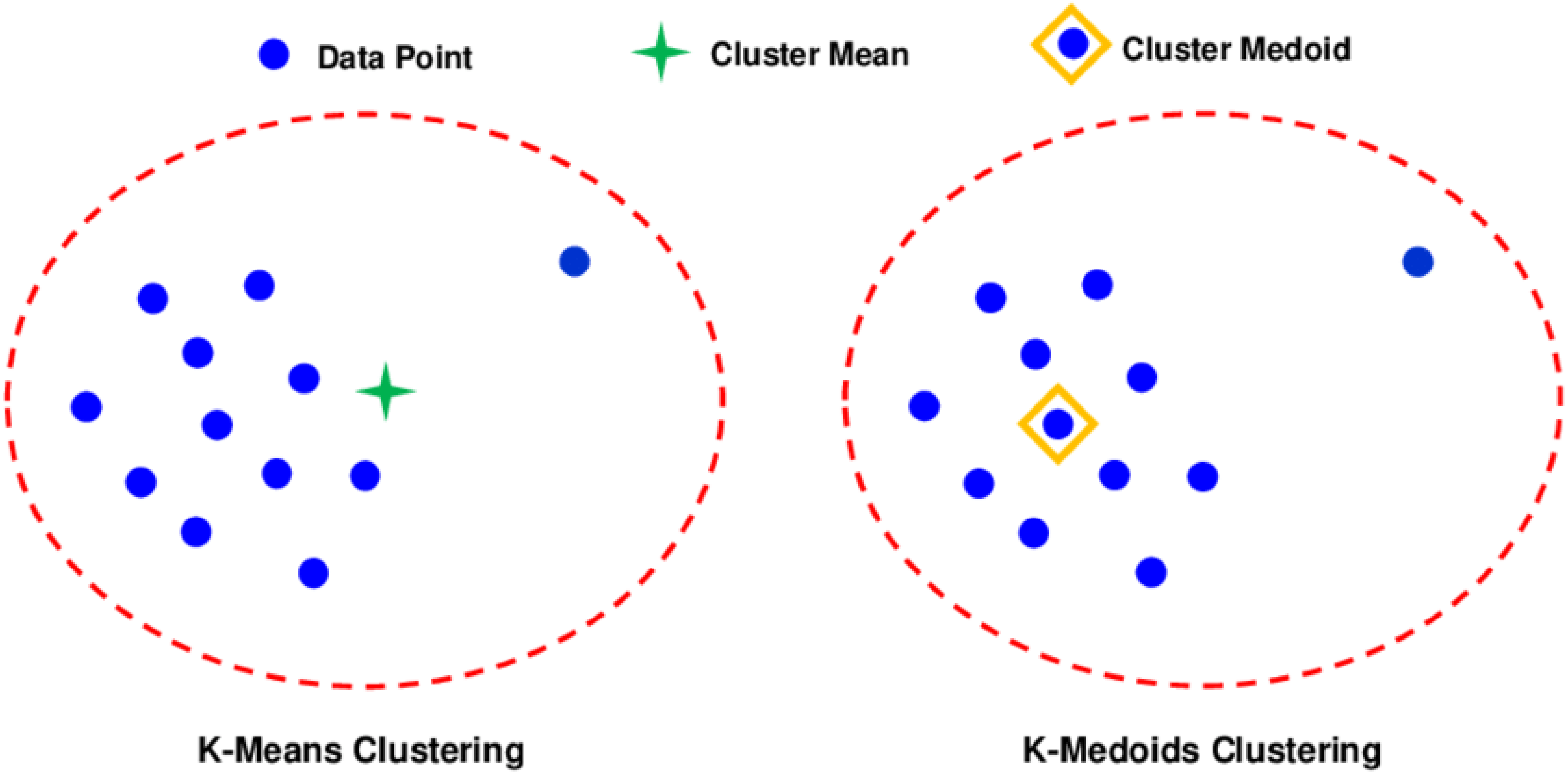

The Choice of Centriod in K-Means is random and what if there is presence of Outlier?

K-Medoids solves the issue as it K-Medoids selected the precise centroid among all data points of the corresponding clusters

https://www.kaggle.com/getting-started/172928

# Unsupervised Learning – K-Means and K-Medoids

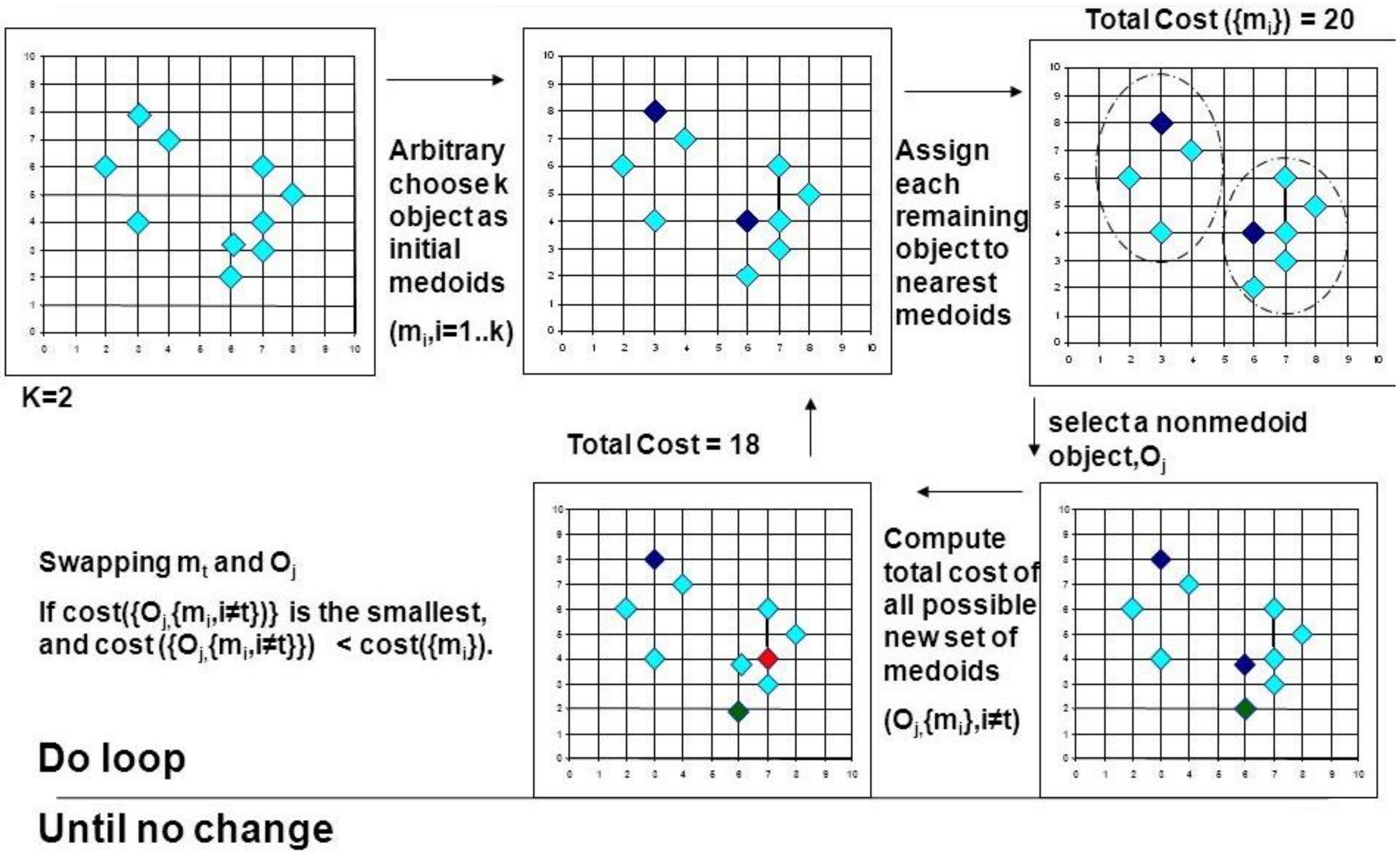





https://www.researchgate.net/figure/The-graphical-representation-of-the-difference-between-the-k-means-and-k-medoids_fig1_342871651

# Unsupervised Learning – K-Means and K-Medoids






https://alfredo-reyes-montero.gitbook.io/soft-computing/applications/matlab/kmeans-kmenoids

# Implementing K-Means



## Creating the dataset

```
from sklearn.datasets import make_blobs
```

```
data = make_blobs(n_samples = 200, n_features = 2, centers = 4, cluster_std = 1.8, random_state = 2022)
```

4 groups

## Defining X and y

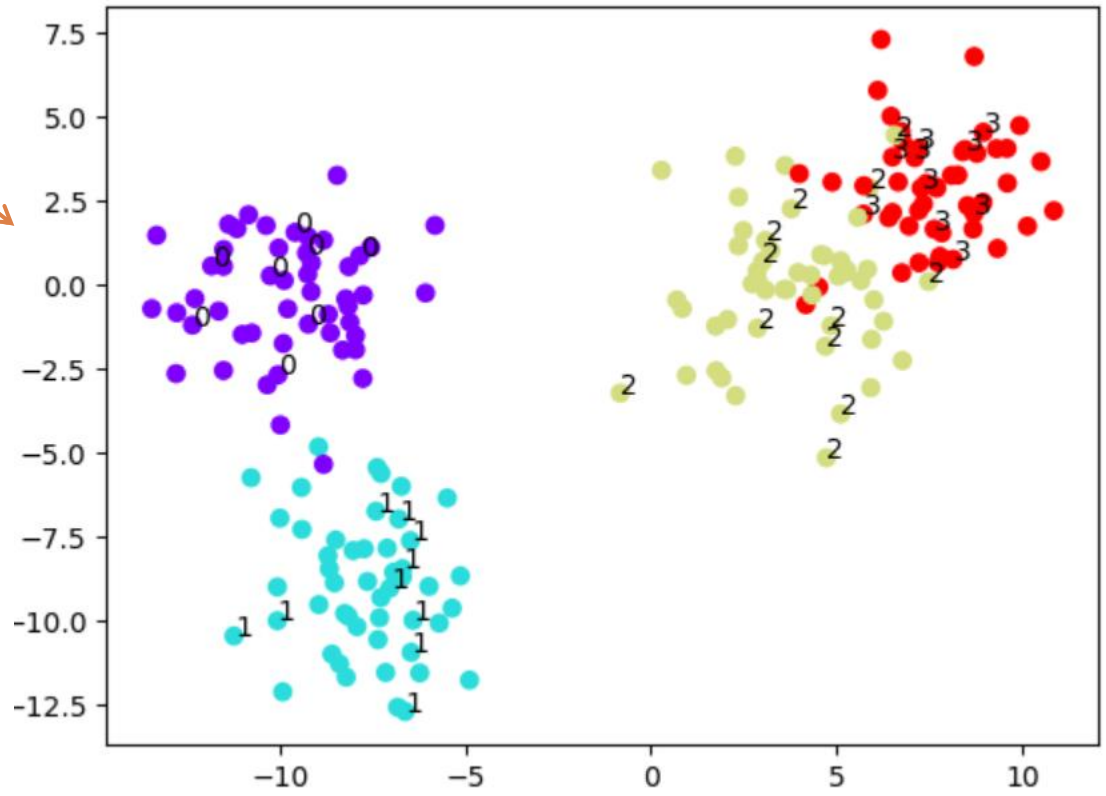
```
X = data[0]
y = data[1]
```

## Visualizing the data (first 5 instances)

```
print('X:', X[0:5, :])
print('Y:', y[0:5])
```

```
X: [[  5.88508997   2.9021639 ]
 [ -8.20429992 -11.68670283]
 [  1.9125188   -2.76746603]
 [ -9.39601207  -7.2830252 ]
 [  6.1986976    7.32152342]]
Y: [2 1 2 1 3]
```

```
plt.scatter(X[:, 0], X[:, 1], c = y, cmap = 'rainbow')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i, 0], X[i, 1]))
```



# Implementing K-Means



sklearn.cluster.KMeans(n_clusters=8, *, init='k-means++', n_init='auto', max_iter=300, tol=0.0001, verbose=0, random_state=None, copy_x=True, algorithm='lloyd')

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

Creating the clusters

```
kmeans = KMeans(n_clusters = 4, n_init = 10, random_state = 2022)
```

```
kmeans.fit(X)
```

```
KMeans
KMeans(n_clusters=4, n_init=10, random_state=2022)
```

Vector with center of the clusters

```
kmeans.cluster_centers_
```

```
array([[-7.68797564, -8.88054369],
       [ 7.70499062,  2.96975295],
       [-9.78544862, -0.2509739 ],
       [ 3.60428123, -0.21752545]])
```

```
kmeans.labels_
```

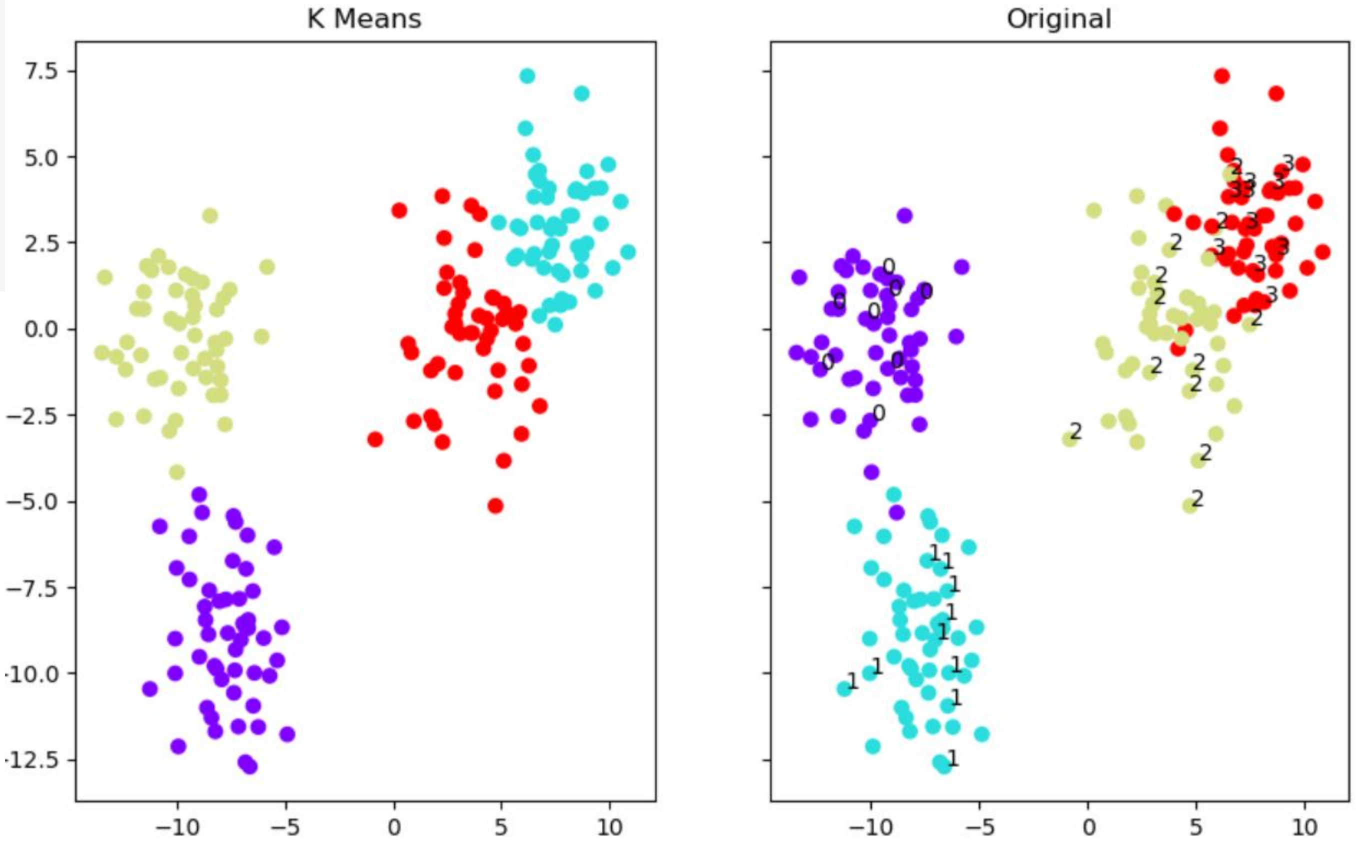
```
array([1, 0, 3, 0, 1, 2, 0, 2, 3, 3, 1, 1, 3, 1, 2, 1, 1, 1, 1, 2, 0, 3,
       1, 1, 0, 0, 3, 0, 3, 3, 1, 0, 1, 3, 1, 3, 0, 1, 0, 2, 1, 2, 1, 3,
       3, 3, 0, 0, 3, 0, 0, 3, 1, 1, 0, 3, 1, 3, 1, 2, 2, 2, 3, 3, 1, 2,
       0, 2, 2, 2, 3, 2, 3, 0, 0, 2, 2, 0, 0, 2, 1, 1, 2, 3, 0, 0, 1, 3,
       2, 0, 3, 1, 2, 3, 3, 1, 3, 2, 3, 3, 0, 1, 2, 0, 2, 3, 2, 3, 2, 2,
       1, 2, 2, 2, 3, 2, 3, 3, 0, 3, 1, 1, 1, 3, 0, 0, 1, 3, 2, 1, 0, 2,
       2, 0, 0, 0, 2, 2, 1, 0, 0, 0, 1, 3, 0, 1, 0, 2, 3, 2, 3, 3, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 0, 0, 3, 1, 2, 0, 0, 1, 2, 3, 1, 1, 0, 3, 1,
       3, 0, 0, 2, 3, 0, 2, 2, 0, 2, 1, 1, 2, 3, 2, 3, 3, 0, 2, 1, 2, 0,
       2, 2])
```

# Implementing K-Means



Visualizing the clusters

```
f, (ax1, ax2) = plt.subplots(1, 2, sharey = True, figsize = (10, 6))
ax1.set_title('K Means')
ax1.scatter(X[:, 0], X[:, 1], c = kmeans.labels_, cmap = 'rainbow')
ax2.set_title("Original")
ax2.scatter(X[:, 0], X[:, 1], c = y, cmap = 'rainbow')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i, 0], X[i, 1]))
```




Note that the colors are meaningless when in reference to the two plots.

# Implementing K-Means



## Align K-Means prediction class with real values

```
y_pred = kmeans.predict(X)
```

```
y_pred
```

```
array([1, 0, 3, 0, 1, 2, 0, 2, 3, 3, 1, 1, 3, 1, 2, 1, 1, 1, 1, 2, 0, 3,
       1, 1, 0, 0, 3, 0, 3, 3, 1, 0, 1, 3, 1, 3, 0, 1, 0, 2, 1, 2, 1, 3,
       3, 3, 0, 0, 3, 0, 0, 3, 1, 1, 0, 3, 1, 3, 1, 2, 2, 2, 3, 3, 1, 2,
       0, 2, 2, 2, 3, 2, 3, 0, 0, 2, 2, 0, 0, 2, 1, 1, 2, 3, 0, 0, 1, 3,
       2, 0, 3, 1, 2, 3, 3, 1, 3, 2, 3, 3, 0, 1, 2, 0, 2, 3, 2, 3, 2, 2,
       1, 2, 2, 2, 3, 2, 3, 3, 0, 3, 1, 1, 1, 3, 0, 0, 1, 3, 2, 1, 0, 2,
       2, 0, 0, 0, 2, 2, 1, 0, 0, 0, 1, 3, 0, 1, 0, 2, 3, 2, 3, 3, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 0, 0, 3, 1, 2, 0, 0, 1, 2, 3, 1, 1, 0, 3, 1,
       3, 0, 0, 2, 3, 0, 2, 2, 0, 2, 1, 1, 2, 3, 2, 3, 3, 0, 2, 1, 2, 0,
       2, 2])
```

```
y
```

```
array([2, 1, 2, 1, 3, 0, 1, 0, 2, 2, 3, 3, 2, 3, 0, 3, 3, 3, 3, 0, 1, 2,
       3, 3, 1, 1, 3, 1, 2, 2, 3, 1, 3, 3, 3, 2, 1, 3, 1, 0, 3, 0, 3, 2,
       2, 2, 1, 1, 2, 1, 1, 2, 3, 3, 1, 2, 3, 2, 3, 0, 0, 0, 2, 2, 3, 0,
       1, 0, 0, 0, 2, 0, 2, 1, 1, 0, 0, 1, 1, 0, 2, 3, 0, 2, 1, 1, 3, 2,
       0, 1, 2, 3, 0, 2, 2, 3, 2, 0, 2, 2, 1, 3, 0, 1, 0, 2, 0, 2, 0, 0,
       3, 0, 0, 0, 2, 0, 2, 2, 1, 2, 3, 3, 3, 2, 1, 1, 3, 2, 0, 3, 1, 0,
       0, 1, 1, 1, 0, 0, 3, 1, 1, 1, 3, 2, 1, 3, 1, 0, 2, 0, 2, 2, 3, 3,
       2, 3, 3, 3, 0, 1, 1, 1, 1, 2, 3, 0, 1, 1, 3, 0, 2, 3, 3, 1, 2, 3,
       2, 1, 1, 0, 2, 1, 0, 0, 1, 0, 3, 3, 0, 2, 0, 3, 2, 1, 0, 2, 0, 1,
       0, 0])
```

```
y_pred = np.where(y_pred==0, 10, y_pred)
y_pred = np.where(y_pred==2, 0, y_pred)
y_pred = np.where(y_pred==3, 2, y_pred)
y_pred = np.where(y_pred==1, 3, y_pred)
y_pred = np.where(y_pred==10, 1, y_pred)
```

```
y_pred
```

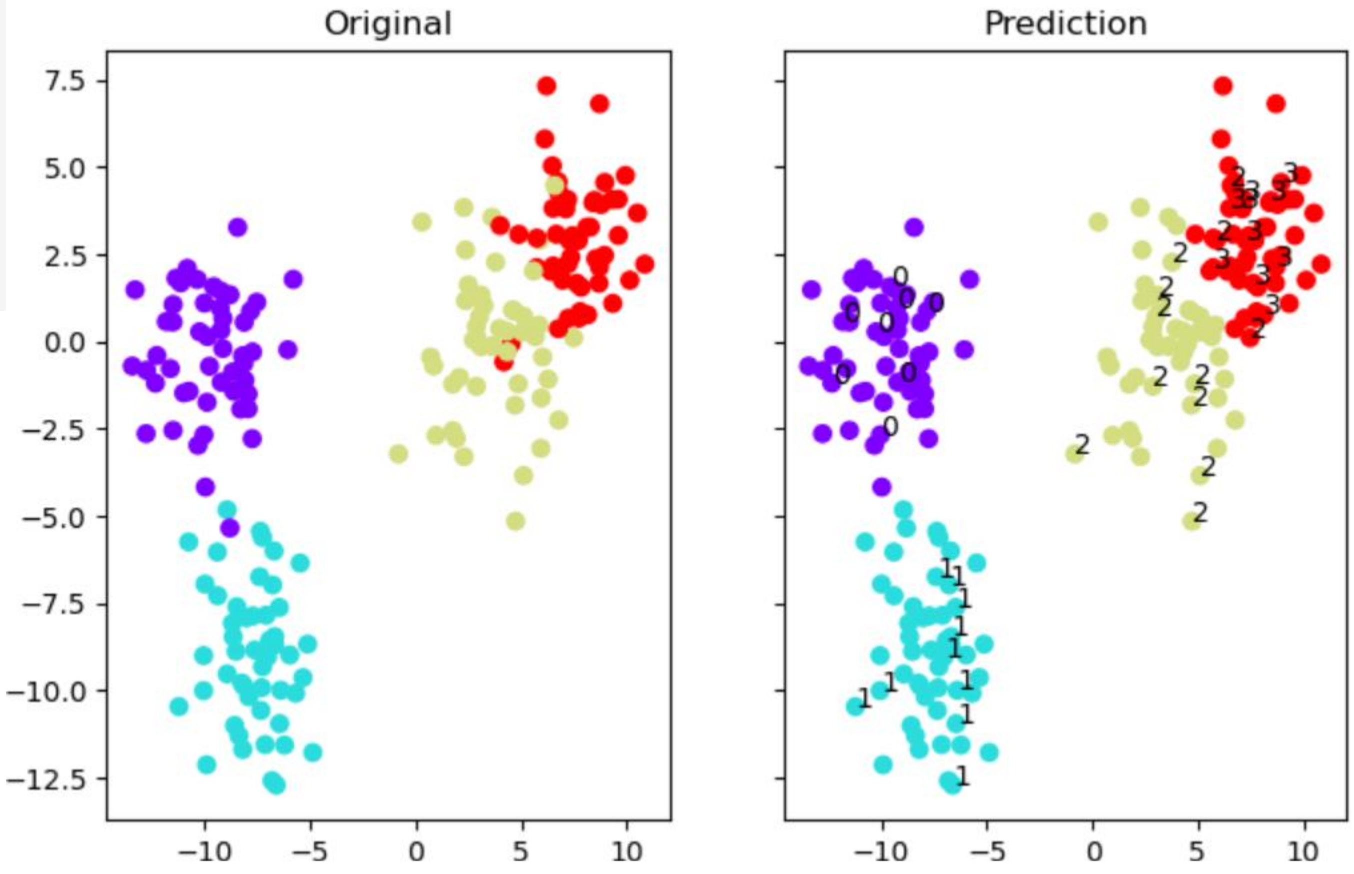
```
array([3, 1, 2, 1, 3, 0, 1, 0, 2, 2, 3, 3, 2, 3, 0, 3, 3, 3, 3, 0, 1, 2,
       3, 3, 1, 1, 2, 1, 2, 2, 3, 1, 3, 2, 3, 2, 1, 3, 1, 0, 3, 0, 3, 2,
       2, 2, 1, 1, 2, 1, 1, 2, 3, 3, 1, 2, 3, 2, 3, 0, 0, 0, 2, 2, 3, 0,
       1, 0, 0, 0, 2, 0, 2, 1, 1, 0, 0, 1, 1, 0, 3, 3, 0, 2, 1, 1, 3, 2,
       0, 1, 2, 3, 0, 2, 2, 3, 2, 0, 2, 2, 1, 3, 0, 1, 0, 2, 0, 2, 0, 0,
       3, 0, 0, 0, 2, 0, 2, 2, 1, 2, 3, 3, 3, 2, 1, 1, 3, 2, 0, 3, 1, 0,
       0, 1, 1, 1, 0, 0, 3, 1, 1, 1, 3, 2, 1, 3, 1, 0, 2, 0, 2, 2, 3, 3,
       3, 3, 3, 3, 1, 1, 1, 1, 1, 2, 3, 0, 1, 1, 3, 0, 2, 3, 3, 1, 2, 3,
       2, 1, 1, 0, 2, 1, 0, 0, 1, 0, 3, 3, 0, 2, 0, 2, 2, 1, 0, 3, 0, 1,
       0, 0])
```

# Implementing K-Means



Redo the visualizations

```
f, (ax1, ax2) = plt.subplots(1, 2, sharey = True, figsize = (8, 5))
ax1.set_title('Original')
ax1.scatter(X[:,0], X[:,1], c=y, cmap='rainbow')
ax2.set_title("Prediction")
ax2.scatter(X[:,0], X[:,1], c=y_pred, cmap='rainbow')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i,0], X[i,1]))
plt.savefig("KMeans_pred.png")
```



# Implementing K-Means



Evaluate the model

```
print(confusion_matrix(y, y_pred))
```

```
[[49  1  0  0]
 [ 0 50  0  0]
 [ 0  0 46  4]
 [ 0  0  3 47]]
```

```
print(classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        50
           1       0.98      1.00      0.99        50
           2       0.94      0.92      0.93        50
           3       0.92      0.94      0.93        50

    accuracy                           0.96       200
   macro avg       0.96      0.96      0.96       200
weighted avg       0.96      0.96      0.96       200
```

If you want to classify new points, it is best to train a classifier on your clustering result.

# Implementing K-Medoids



Let's use the dataset created earlier.

[sklearn_extra.cluster.KMedoids(n_clusters=8, metric='euclidean', method='alternate', init='heuristic', max_iter=300, random_state=None)]

```
from sklearn_extra.cluster import KMedoids
```

Creating the clusters

```
kmedoids = KMedoids(n_clusters = 4, random_state = 2022)
kmedoids.fit(X)
```

```
KMedoids
KMedoids(n_clusters=4, random_state=2022)
```

Vector with center of the clusters

```
kmedoids.cluster_centers_
```

```
array([[-7.62795904, -8.8354951 ],
       [-9.87312876,  0.13931247],
       [ 3.65994783, -0.13260646],
       [ 7.66163195,  2.9333056 ]])
```

```
kmedoids.labels_
```

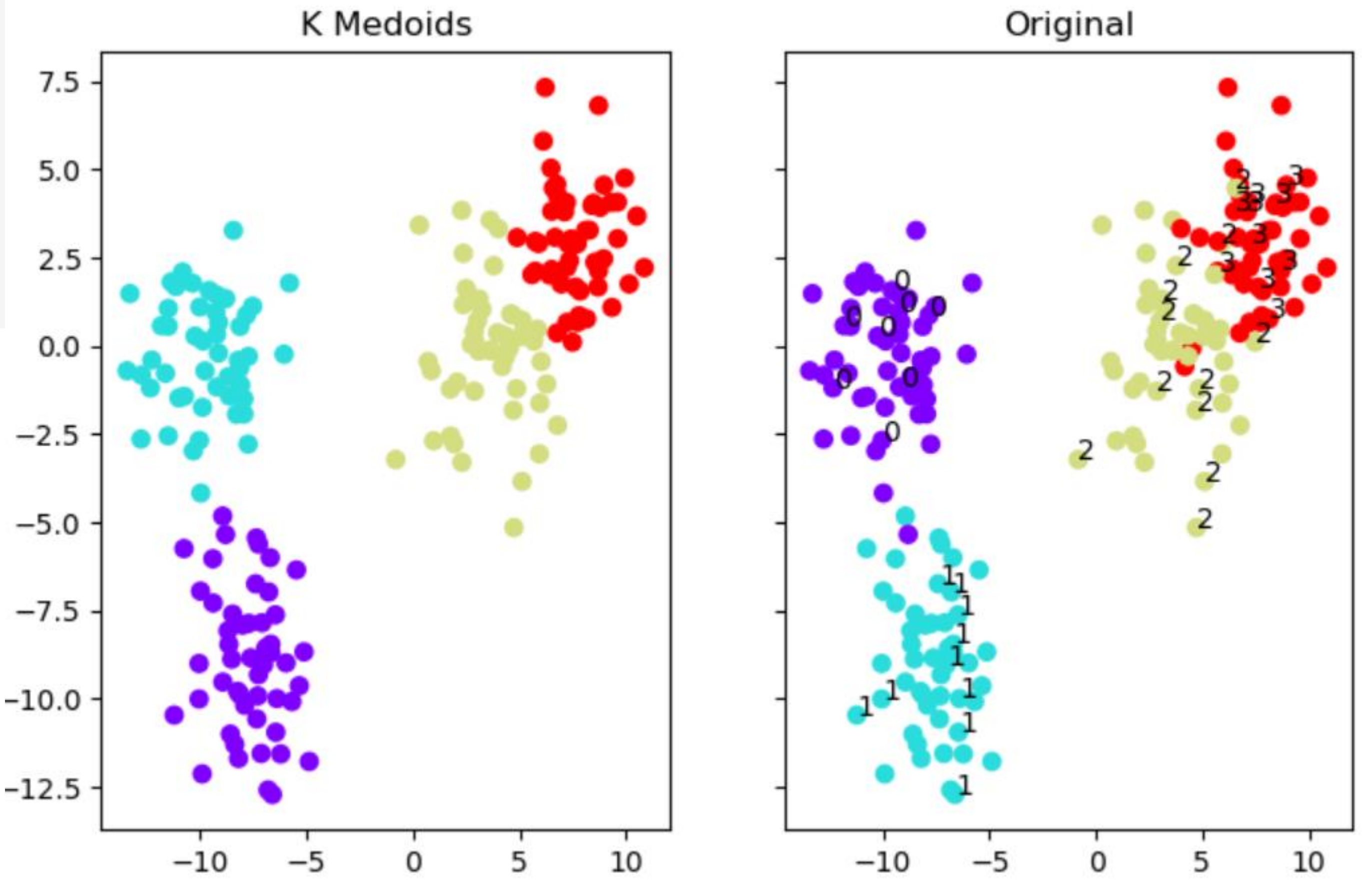
```
array([3, 0, 2, 0, 3, 1, 0, 1, 2, 2, 3, 3, 2, 3, 1, 3, 3, 3, 3, 1, 0, 2,
       3, 3, 0, 0, 2, 0, 2, 2, 3, 0, 3, 2, 3, 2, 0, 3, 0, 1, 3, 1, 3, 2,
       2, 2, 0, 0, 2, 0, 0, 2, 3, 3, 0, 2, 3, 2, 3, 1, 1, 1, 2, 2, 3, 1,
       0, 1, 1, 1, 2, 1, 2, 0, 0, 1, 1, 0, 0, 1, 3, 3, 1, 2, 0, 0, 3, 2,
       1, 0, 2, 3, 1, 2, 2, 3, 2, 1, 2, 2, 0, 3, 1, 0, 1, 2, 1, 2, 1, 1,
       3, 1, 1, 1, 2, 1, 2, 2, 0, 2, 3, 3, 3, 2, 0, 0, 3, 2, 1, 3, 0, 1,
       1, 0, 0, 0, 1, 1, 3, 0, 0, 0, 3, 2, 0, 3, 0, 1, 2, 1, 2, 2, 3, 3,
       3, 3, 3, 3, 0, 0, 0, 0, 0, 2, 3, 1, 0, 0, 3, 1, 2, 3, 3, 0, 2, 3,
       2, 0, 0, 1, 2, 0, 1, 1, 0, 1, 3, 3, 1, 2, 1, 2, 2, 0, 1, 3, 1, 0,
       1, 1], dtype=int64)
```

# Implementing K-Medoids



Visualizing the clusters

```
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True,figsize=(8,5))
ax1.set_title('K Medoids')
ax1.scatter(X[:, 0], X[:, 1], c = kmedoids.labels_, cmap = 'rainbow')
ax2.set_title("Original")
ax2.scatter(X[:, 0], X[:, 1], c = y, cmap = 'rainbow')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i, 0], X[i, 1]))
plt.savefig("KMedoids.png")
```



# Implementing K-Medoids



## Align K-Medoids prediction class with real values

```
y_pred = kmedoids.predict(X)
```

```
y_pred
```

```
array([3, 0, 2, 0, 3, 1, 0, 1, 2, 2, 3, 3, 2, 3, 1, 3, 3, 3, 3, 1, 0, 2,
       3, 3, 0, 0, 2, 0, 2, 2, 3, 0, 3, 2, 3, 2, 0, 3, 0, 1, 3, 1, 3, 2,
       2, 2, 0, 0, 2, 0, 0, 2, 3, 3, 0, 2, 3, 2, 3, 1, 1, 1, 2, 2, 3, 1,
       0, 1, 1, 1, 2, 1, 2, 0, 0, 1, 1, 0, 0, 1, 3, 3, 1, 2, 0, 0, 3, 2,
       1, 0, 2, 3, 1, 2, 2, 3, 2, 1, 2, 2, 0, 3, 1, 0, 1, 2, 1, 2, 1, 1,
       3, 1, 1, 1, 2, 1, 2, 2, 0, 2, 3, 3, 3, 2, 0, 0, 3, 2, 1, 3, 0, 1,
       1, 0, 0, 0, 1, 1, 3, 0, 0, 0, 3, 2, 0, 3, 0, 1, 2, 1, 2, 2, 3, 3,
       3, 3, 3, 3, 0, 0, 0, 0, 0, 2, 3, 1, 0, 0, 3, 1, 2, 3, 3, 0, 2, 3,
       2, 0, 0, 1, 2, 0, 1, 1, 0, 1, 3, 3, 1, 2, 1, 2, 2, 0, 1, 3, 1, 0,
       1, 1], dtype=int64)
```

```
y
```

```
array([2, 1, 2, 1, 3, 0, 1, 0, 2, 2, 3, 3, 2, 3, 0, 3, 3, 3, 3, 0, 1, 2,
       3, 3, 1, 1, 3, 1, 2, 2, 3, 1, 3, 3, 3, 2, 1, 3, 1, 0, 3, 0, 3, 2,
       2, 2, 1, 1, 2, 1, 1, 2, 3, 3, 1, 2, 3, 2, 3, 0, 0, 0, 2, 2, 3, 0,
       1, 0, 0, 0, 2, 0, 2, 1, 1, 0, 0, 1, 1, 0, 2, 3, 0, 2, 1, 1, 3, 2,
       0, 1, 2, 3, 0, 2, 2, 3, 2, 0, 2, 2, 1, 3, 0, 1, 0, 2, 0, 2, 0, 0,
       3, 0, 0, 0, 2, 0, 2, 2, 1, 2, 3, 3, 3, 2, 1, 1, 3, 2, 0, 3, 1, 0,
       0, 1, 1, 1, 0, 0, 3, 1, 1, 1, 3, 2, 1, 3, 1, 0, 2, 0, 2, 2, 3, 3,
       2, 3, 3, 3, 0, 1, 1, 1, 1, 2, 3, 0, 1, 1, 3, 0, 2, 3, 3, 1, 2, 3,
       2, 1, 1, 0, 2, 1, 0, 0, 1, 0, 3, 3, 0, 2, 0, 3, 2, 1, 0, 2, 0, 1,
       0, 0])
```

```
y_pred = np.where(y_pred==1, 10, y_pred)
y_pred = np.where(y_pred==0, 1, y_pred)
y_pred = np.where(y_pred==10, 0, y_pred)
```

```
y_pred
```

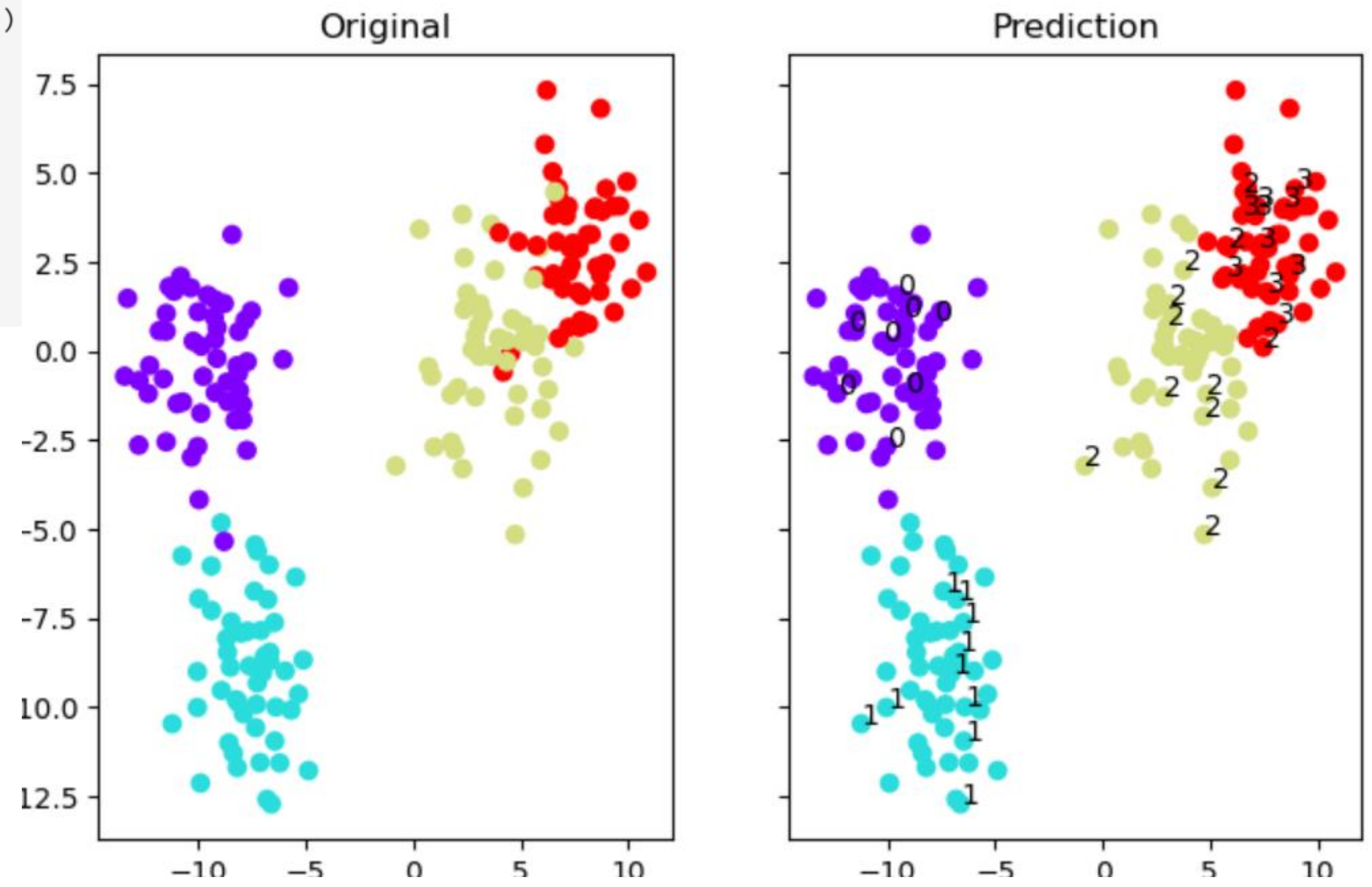
```
array([3, 1, 2, 1, 3, 0, 1, 0, 2, 2, 3, 3, 2, 3, 0, 3, 3, 3, 3, 0, 1, 2,
       3, 3, 1, 1, 2, 1, 2, 2, 3, 1, 3, 2, 3, 2, 1, 3, 1, 0, 3, 0, 3, 2,
       2, 2, 1, 1, 2, 1, 1, 2, 3, 3, 1, 2, 3, 2, 3, 0, 0, 0, 2, 2, 3, 0,
       1, 0, 0, 0, 2, 0, 2, 1, 1, 0, 0, 1, 1, 0, 3, 3, 0, 2, 1, 1, 3, 2,
       0, 1, 2, 3, 0, 2, 2, 3, 2, 0, 2, 2, 1, 3, 0, 1, 0, 2, 0, 2, 0, 0,
       3, 0, 0, 0, 2, 0, 2, 2, 1, 2, 3, 3, 3, 2, 1, 1, 3, 2, 0, 3, 1, 0,
       0, 1, 1, 1, 0, 0, 3, 1, 1, 1, 3, 2, 1, 3, 1, 0, 2, 0, 2, 2, 3, 3,
       3, 3, 3, 3, 1, 1, 1, 1, 1, 2, 3, 0, 1, 1, 3, 0, 2, 3, 3, 1, 2, 3,
       2, 1, 1, 0, 2, 1, 0, 0, 1, 0, 3, 3, 0, 2, 0, 2, 2, 1, 0, 3, 0, 1,
       0, 0], dtype=int64)
```

# Implementing K-Medoids



Redo the visualizations

```
f, (ax1, ax2) = plt.subplots(1, 2, sharey = True, figsize = (8, 5))
ax1.set_title('Original')
ax1.scatter(X[:, 0], X[:, 1], c = y, cmap = 'rainbow')
ax2.set_title("Prediction")
ax2.scatter(X[:, 0], X[:, 1], c = y_pred, cmap = 'rainbow')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i, 0], X[i, 1]))
plt.savefig("KMedoid_pred.png")
```



# Implementing K-Medoids



## Evaluate the model

```
print(confusion_matrix(y, y_pred))
```

```
[[49  1  0  0]
 [ 0 50  0  0]
 [ 0  0 46  4]
 [ 0  0  3 47]]
```

```
print(classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        50
           1       0.98      1.00      0.99        50
           2       0.94      0.92      0.93        50
           3       0.92      0.94      0.93        50

    accuracy                           0.96       200
   macro avg       0.96      0.96      0.96       200
weighted avg       0.96      0.96      0.96       200
```

# K-Means vs. K-Medoids



## K-Means

```
print(confusion_matrix(y, y_pred))
```

```
[[49  1  0  0]
 [ 0 50  0  0]
 [ 0  0 46  4]
 [ 0  0  3 47]]
```

```
print(classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        50
           1       0.98      1.00      0.99        50
           2       0.94      0.92      0.93        50
           3       0.92      0.94      0.93        50

    accuracy                           0.96       200
   macro avg       0.96      0.96      0.96       200
weighted avg       0.96      0.96      0.96       200
```

## K-Medoids

```
print(confusion_matrix(y, y_pred))
```

```
[[49  1  0  0]
 [ 0 50  0  0]
 [ 0  0 46  4]
 [ 0  0  3 47]]
```

```
print(classification_report(y, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      0.98      0.99        50
           1       0.98      1.00      0.99        50
           2       0.94      0.92      0.93        50
           3       0.92      0.94      0.93        50

    accuracy                           0.96       200
   macro avg       0.96      0.96      0.96       200
weighted avg       0.96      0.96      0.96       200
```

**Can we compare these two models?**

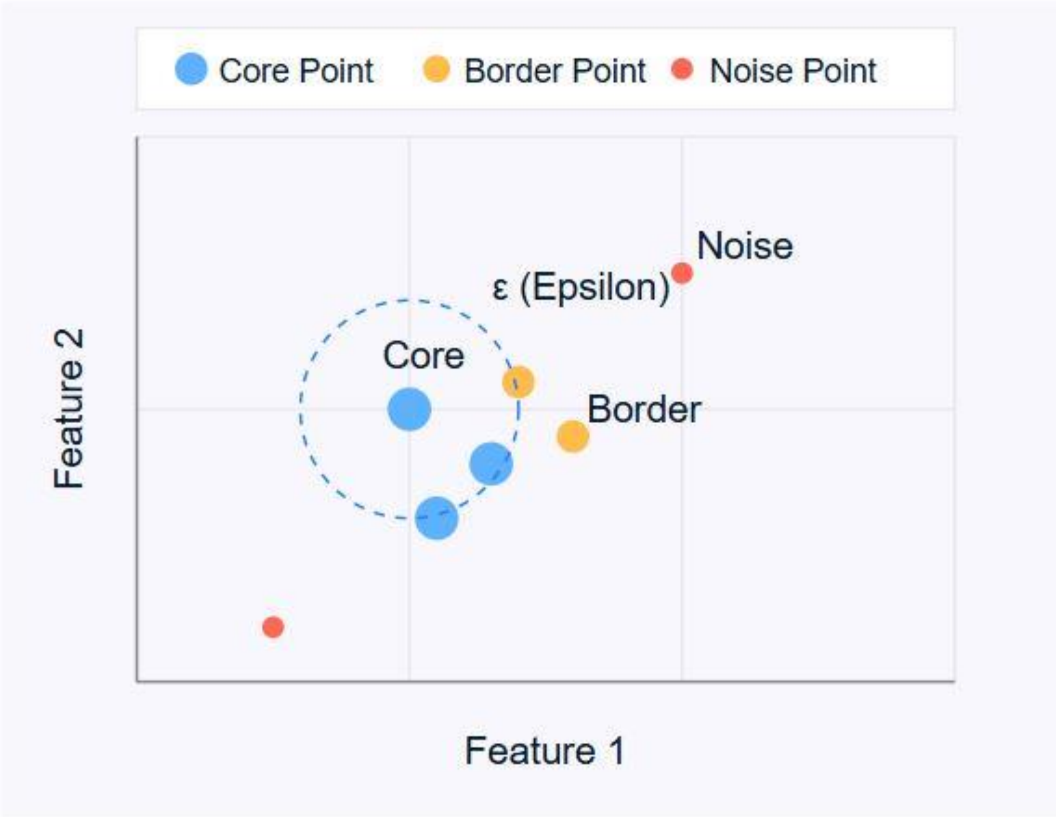
# Unsupervised Learning - DBSCAN



The **DBSCAN** (Density-Based Spatial Clustering of Applications with Noise) algorithm finds core samples in high-density regions and expands clusters for them.

**High-density regions**, where data points are located close to each other, are separated by **low-density regions**, where the data points are located far from each other.

The idea of **a core sample** means a sample located in an area of high-density. Data point A is considered a core sample if at least minimum number of points required to form a dense region (usually including A) are located within ε distance from A.

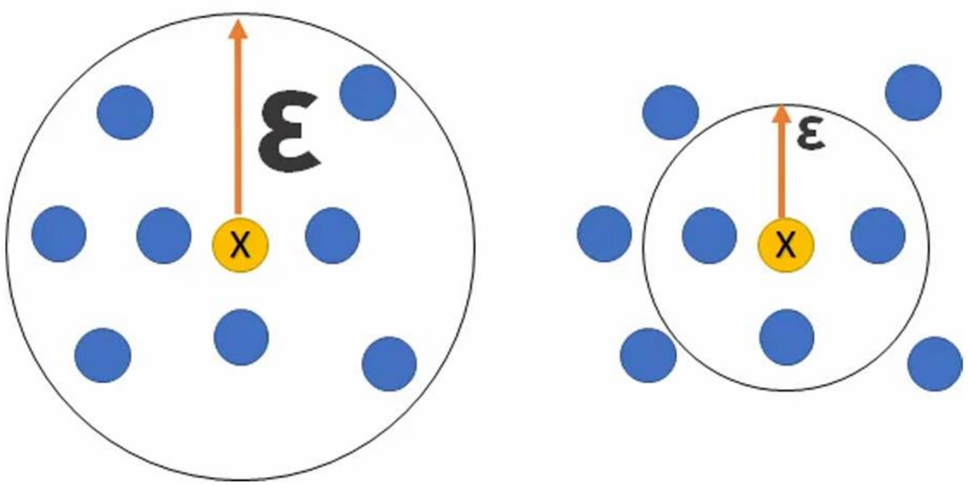



https://www.datacamp.com/tutorial/dbscan-clustering-algorithm

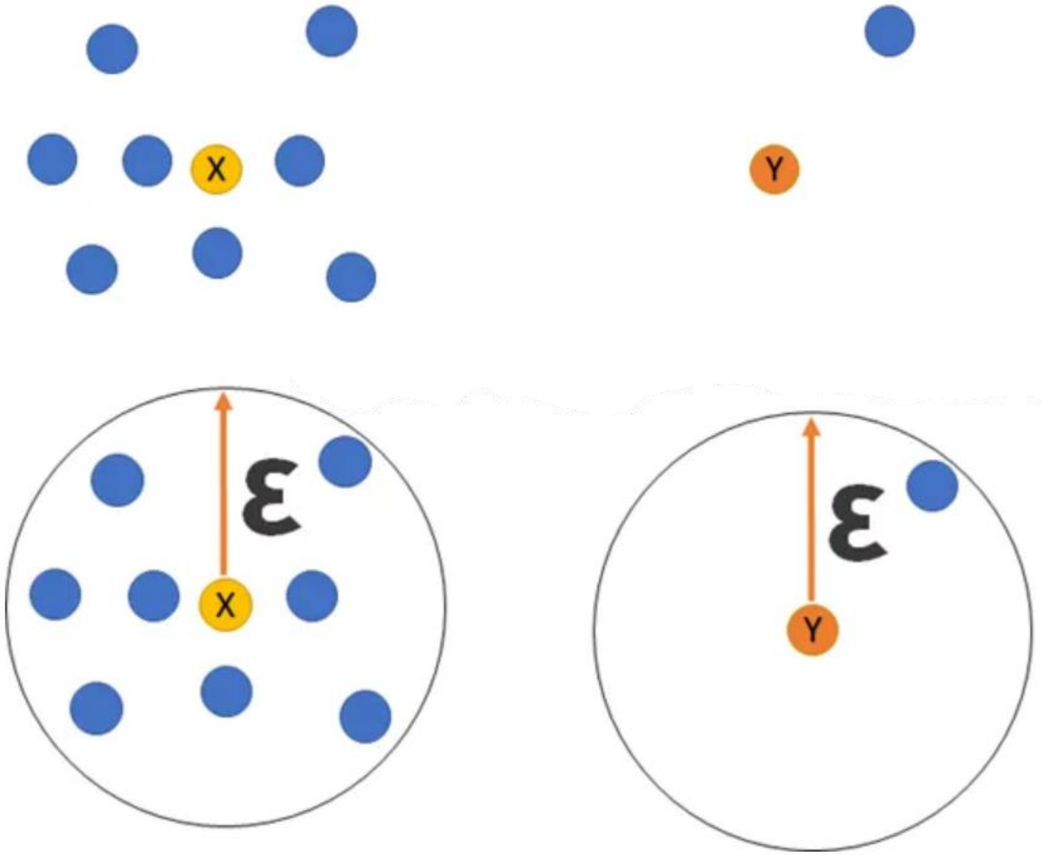
# Unsupervised Learning - DBSCAN



## Influence of the size of $\varepsilon$




## Influence of the density of the region




https://towardsdatascience.com/a-practical-guide-to-dbscan-method-d4ec5ab2bc99

# Unsupervised Learning - DBSCAN



Strengths:

- knowledge about the number of clusters is not necessary;
- also solves the anomaly detection task.

Weaknesses:

- need to select and tune the density parameter ($\varepsilon$);
- does not cope well with sparse data.

# Implementing DBSCAN



**Creating the dataset**

```
X, y = make_blobs(n_samples=750, cluster_std=0.4, random_state=0)
```

3 groups

Shape of X and y and visualizing the data

```
X.shape
```

```
(750, 2)
```

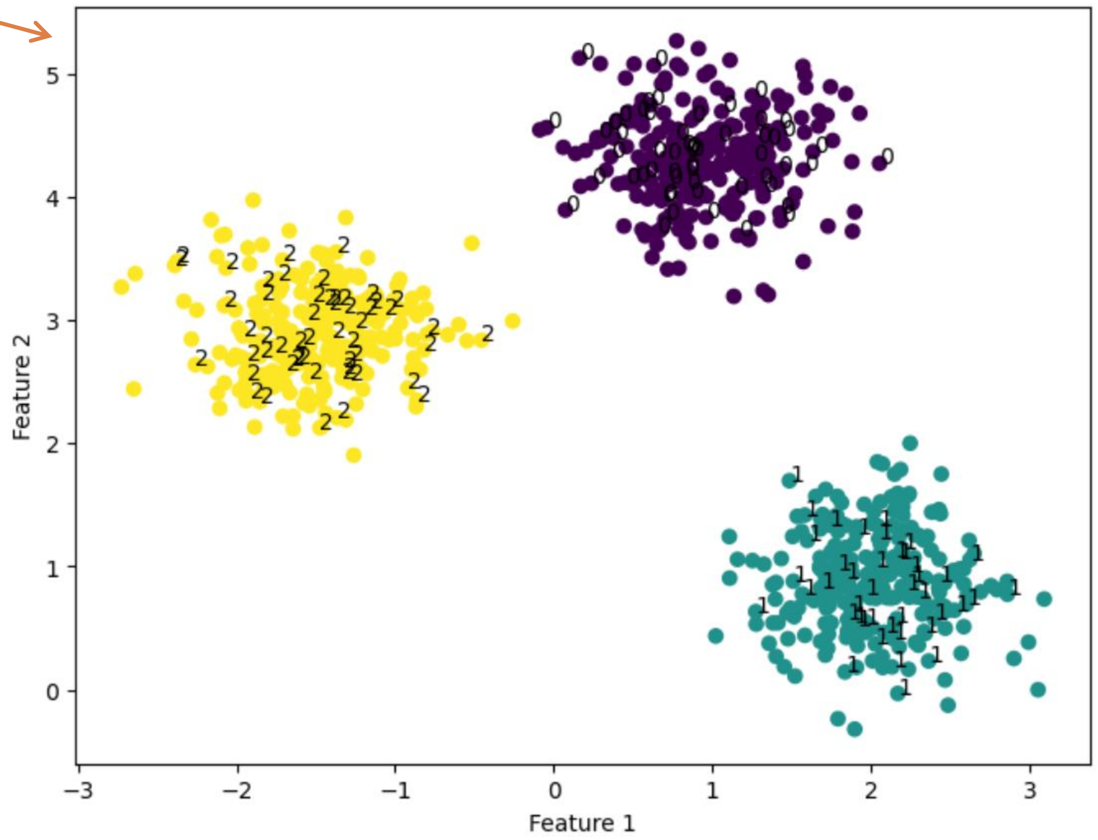
```
y.shape
```

```
(750,)
```

```
print('X:', X[0:5])
print('y:', y[0:5])
```

```
X: [[ 2.36434546  0.23302434]
 [ 0.92311785  4.18467098]
 [ 1.64221028  0.72296432]
 [ 1.97590796  0.93534058]
 [-1.68752703  2.73049184]]
y: [1 0 1 1 2]
```

```
plt.figure(figsize=(8, 6))
plt.scatter(X[:, 0], X[:, 1], c = y, cmap = 'viridis')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i, 0], X[i, 1]))
plt.title('Blobs Dataset')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



# Implementing DBSCAN



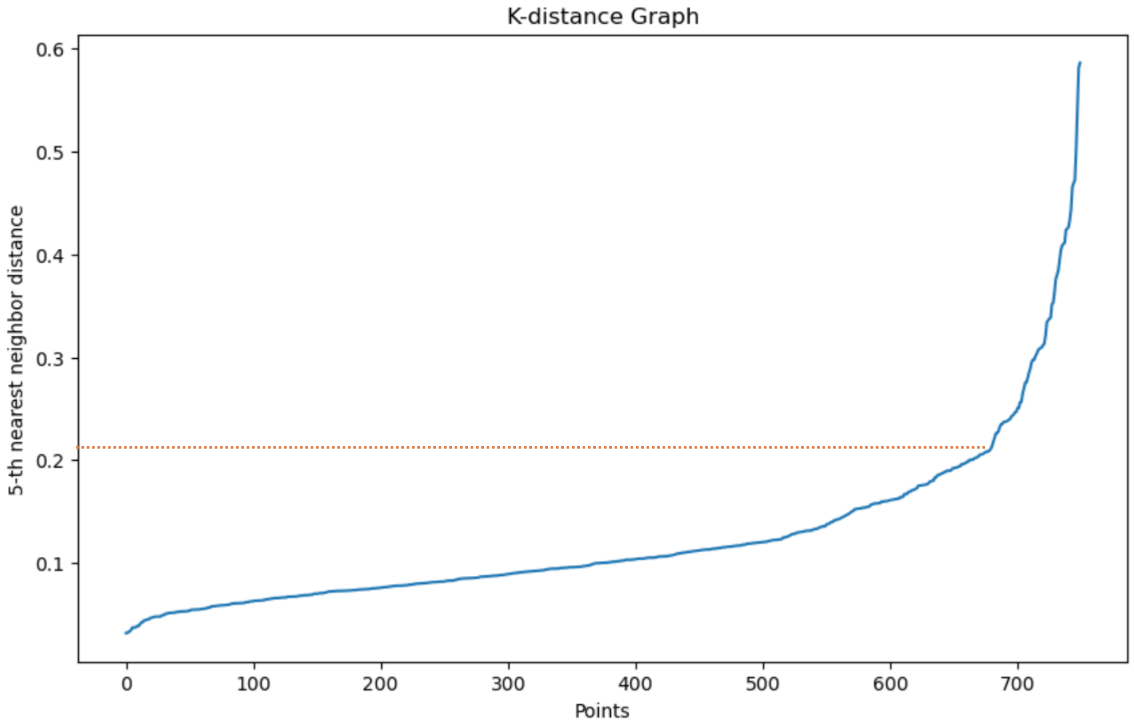
**Determining the epsilon parameter**

We use the *k*-distance graph method to help choose an appropriate $\varepsilon$ value:

1. We define a function `plot_k_distance_graph` that calculates the distance to the $k^{th}$ nearest neighbor for each point
2. The distances are sorted and plotted.
3. We look for an "elbow" in the resulting graph to choose $\varepsilon$.

```
from sklearn.neighbors import NearestNeighbors

def plot_k_distance_graph(X, k):
    neigh = NearestNeighbors(n_neighbors=k)
    neigh.fit(X)
    distances, _ = neigh.kneighbors(X)
    distances = np.sort(distances[:, k-1])
    plt.figure(figsize=(10, 6))
    plt.plot(distances)
    plt.xlabel('Points')
    plt.ylabel(f'{k}-th nearest neighbor distance')
    plt.title('K-distance Graph')
    plt.show()
plot_k_distance_graph(X, k=5)
```



# Implementing DBSCAN



sklearn.cluster.DBSCAN(eps=0.5, *, min_samples=5, metric='euclidean', metric_params=None, algorithm='auto', leaf_size=30, p=None, n_jobs=None)

```
from sklearn.cluster import DBSCAN
```

## Creating the clusters

```
db = DBSCAN(eps=0.2, min_samples=10)
```

```
db.fit(X)
```

```
DBSCAN
DBSCAN(eps=0.2, min_samples=10)
```

Confirm the labels shape

```
labels = db.labels_
labels.shape
```

```
(750,)
```

# Implementing DBSCAN



## Interpreting the results

Indices of core samples

```
db.core_sample_indices_
```

```
array([  1,    2,    3,    4,    6,    8,   10,   12,   14,   15,   16,   18,   20,
       ...,
       736, 738, 739, 740, 741, 742, 743, 745, 746, 748, 749], dtype=int64)
```

Components

```
db.components_
```

```
array([[0.92311785, 4.18467098],
       [1.64221028, 0.72296432],
       [1.97590796, 0.93534058],
       ...,
       [1.04724054, 4.14307495],
       [2.16897939, 1.59473117],
       [0.73774446, 4.28276041]])
```

Number of features

```
db.n_features_in_
```

```
2
```

Number of clusters and noise points in *labels*

```
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
n_noise_ = list(labels).count(-1)
```

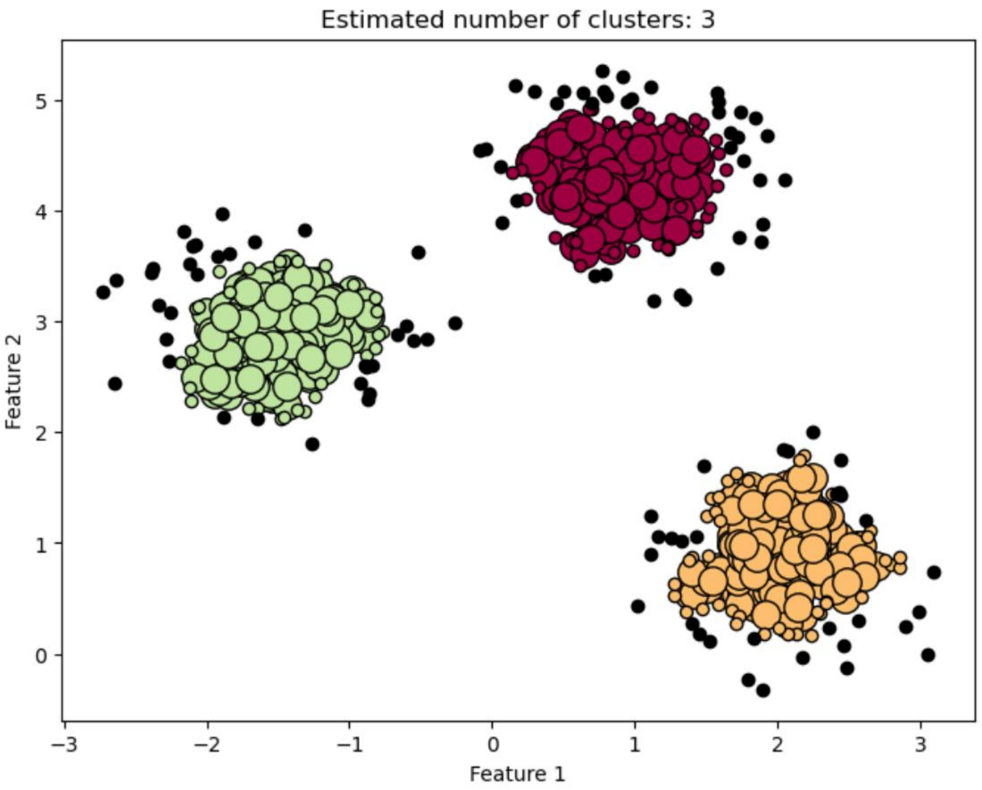
```
print("Estimated number of clusters: %d" % n_clusters_)
print("Estimated number of noise points: %d" % n_noise_)
```

```
Estimated number of clusters: 3
Estimated number of noise points: 104
```

# Implementing DBSCAN



## Visualizing the resultant clusters



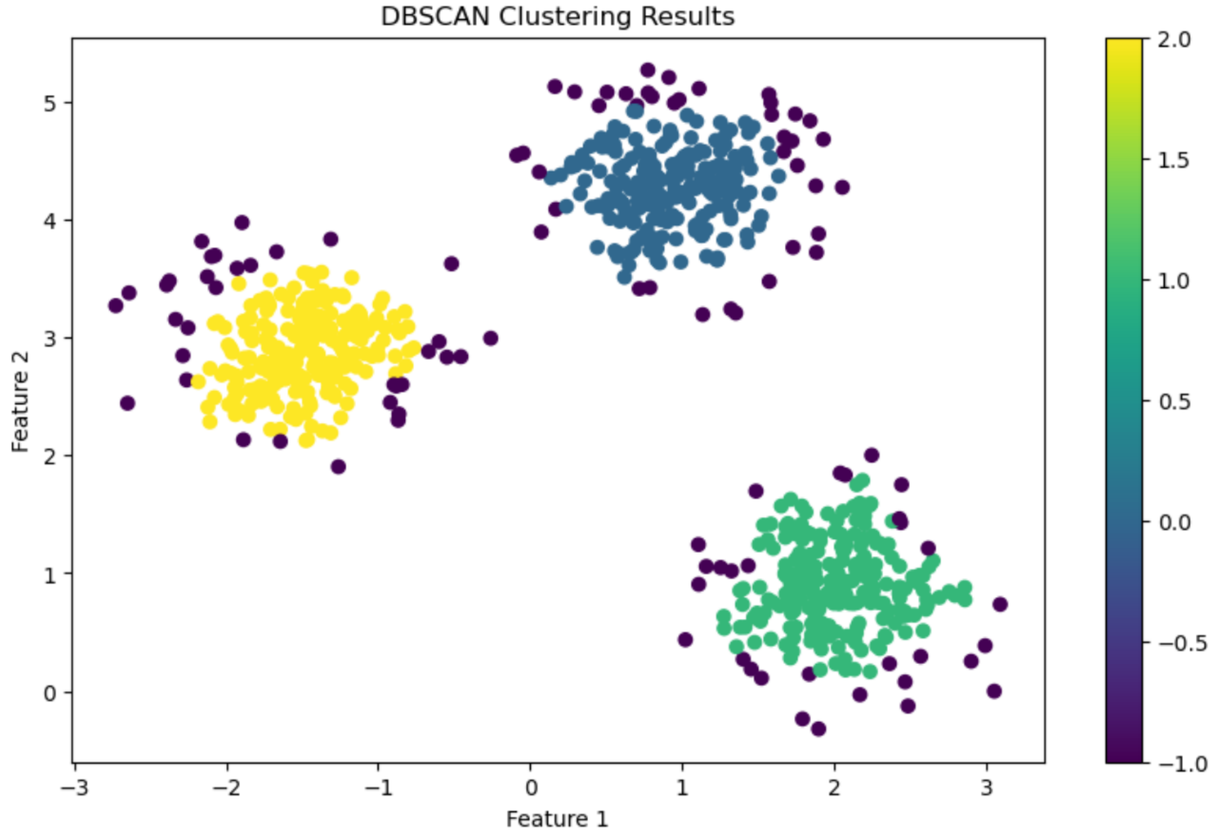

```python
unique_labels = set(labels)
core_samples_mask = np.zeros_like(labels, dtype=bool)
core_samples_mask[db.core_sample_indices_] = True

plt.figure(figsize=(8, 6))
colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]
for k, col in zip(unique_labels, colors):
    if k == -1:
        col = [0, 0, 0, 1]
    class_member_mask = labels == k
    xy = X[class_member_mask & core_samples_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=14,)
    xy = X[class_member_mask & ~core_samples_mask]
    plt.plot(
        xy[:, 0],
        xy[:, 1],
        "o",
        markerfacecolor=tuple(col),
        markeredgecolor="k",
        markersize=6,)
plt.title(f"Estimated number of clusters: {n_clusters_}")
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

# Implementing DBSCAN



```
plt.figure(figsize=(10, 6))
scatter = plt.scatter(X[:, 0], X[:, 1], c=pred, cmap='viridis')
plt.colorbar(scatter)
plt.title('DBSCAN Clustering Results')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



# Implementing DBSCAN



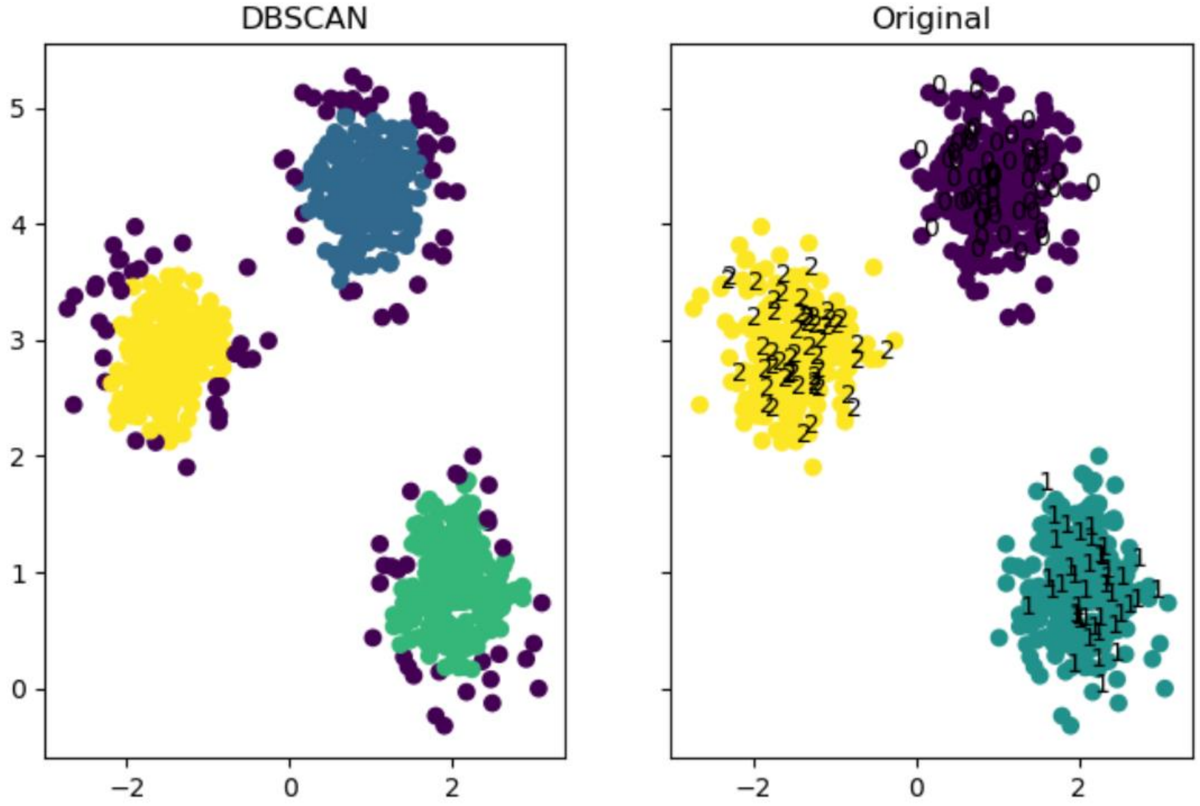
**Evaluating the model**

Obtain the predictions

```
pred = db.fit_predict(X)
```

Visualize the original vs. DBSCAN clusters

```
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True,figsize=(8,5))
ax1.set_title('DBSCAN')
ax1.scatter(X[:, 0], X[:, 1], c = db.labels_, cmap = 'viridis')
ax2.set_title("Original")
ax2.scatter(X[:, 0], X[:, 1], c = y, cmap = 'viridis')
for i, txt in enumerate(y):
    if i%5 == 0:
        plt.annotate(txt, (X[i, 0], X[i, 1]))
plt.savefig("DBSCAN.png")
plt.show()
```



# Implementing DBSCAN



Because Blobs provides the true labels, we can analyze through the following metrics: homogeneity, completeness, V-measure, Adjusted Rand Index, Adjusted Mutual Information and Silhouette Coefficient

```python
print(f"Homogeneity: {metrics.homogeneity_score(y, labels):.3f}")
print(f"Completeness: {metrics.completeness_score(y, labels):.3f}")
print(f"V-measure: {metrics.v_measure_score(y, labels):.3f}")
print(f"Adjusted Rand Index: {metrics.adjusted_rand_score(y, labels):.3f}")
print(
    "Adjusted Mutual Information:"
    f" {metrics.adjusted_mutual_info_score(y, labels):.3f}"
)
print(f"Silhouette Coefficient: {metrics.silhouette_score(X, labels):.3f}")
```

```
Homogeneity: 0.862
Completeness: 0.702
V-measure: 0.774
Adjusted Rand Index: 0.781
Adjusted Mutual Information: 0.773
Silhouette Coefficient: 0.586
```

```python
print(confusion_matrix(y, pred))
```

```
[[  0   0   0   0]
 [ 39 211   0   0]
 [ 31   0 219   0]
 [ 34   0   0 216]]
```

```python
print(classification_report(y, pred))
```

```
              precision    recall  f1-score   support

          -1       0.00      0.00      0.00         0
           0       1.00      0.84      0.92       250
           1       1.00      0.88      0.93       250
           2       1.00      0.86      0.93       250

    accuracy                           0.86       750
   macro avg       0.75      0.65      0.69       750
weighted avg       1.00      0.86      0.93       750
```

What do these metrics mean?

# Implementing DBSCAN



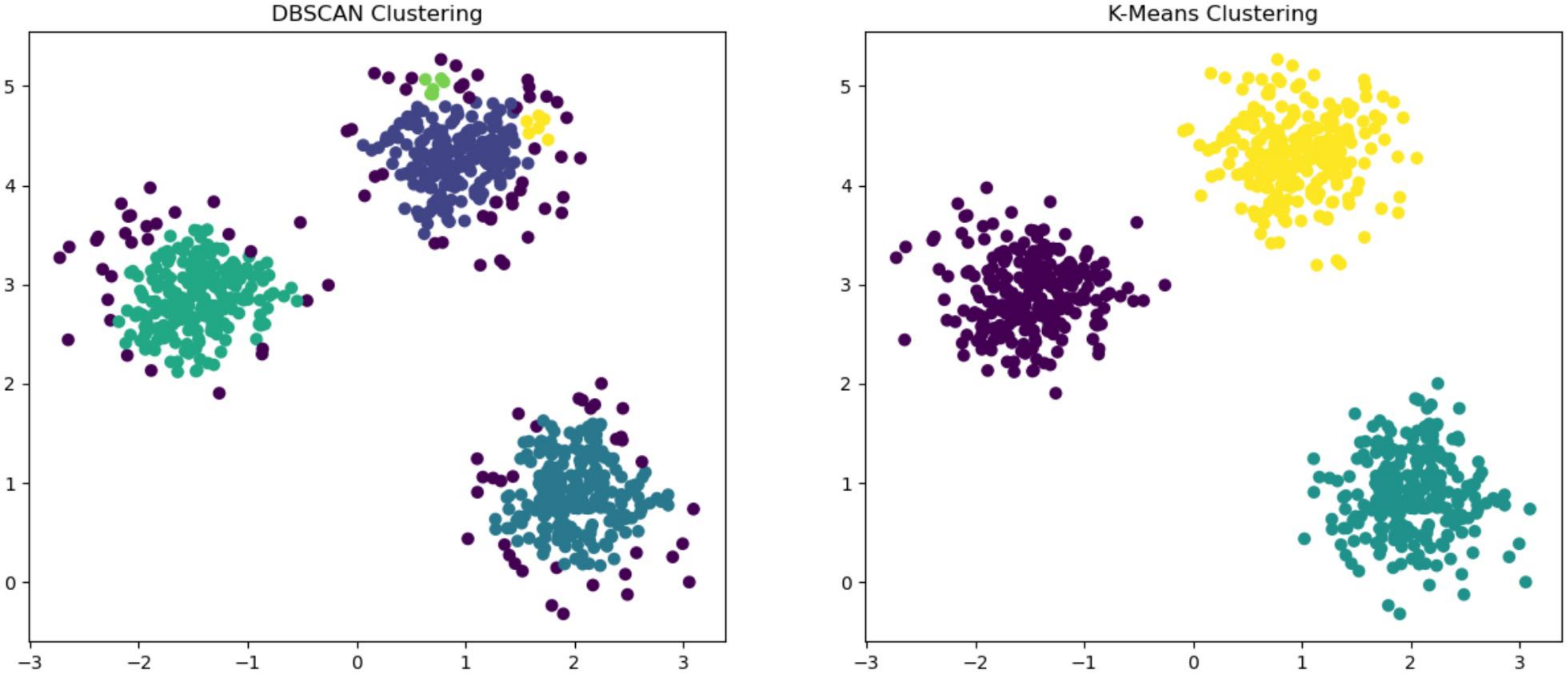
**Comparing DBSCAN and K-Means**

```
dbscan = DBSCAN(eps=0.15, min_samples=5)
dbscan_labels = dbscan.fit_predict(X)
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))
ax1.scatter(X[:, 0], X[:, 1], c=dbscan_labels, cmap='viridis')
ax1.set_title('DBSCAN Clustering')
ax2.scatter(X[:, 0], X[:, 1], c=kmeans_labels, cmap='viridis')
ax2.set_title('K-Means Clustering')
plt.show()
```

# Implementing DBSCAN



## Comparing DBSCAN and K-Means




What conclusions can be obtained?

# Hands On