Unsupervised Learning - Part 1

Machine Learning for Engineering Applications

Fall 2023

K-Means Clustering

- **k-means** → widely used in academia as well as in industry

- Is it good for Engineering? Keep that in mind…

- **Clustering** → technique that allows to find groups of similar objects, objects that are more related to each other than to objects in other groups.

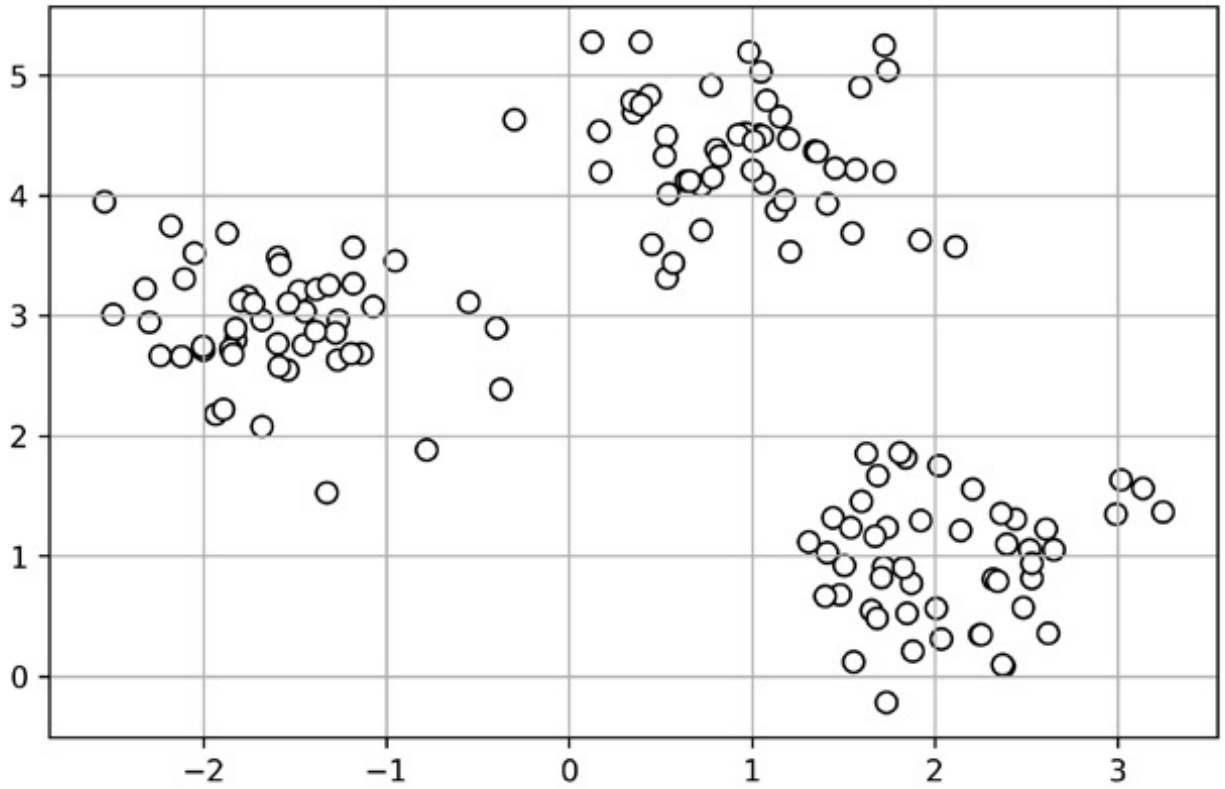
- k-means algorithm
  - easy to implement
  - computationally very efficient

- k-means algorithm → category of **prototype-based clustering**

- **Prototype-based clustering** → each cluster is represented by a prototype either be:
  - **the centroid** (*average*) of similar points with continuous features
  - **the medoid** (*the most representative or most frequently occurring point*) in the case of categorical features.

- Drawback → specify the number of clusters, *k* (*a priori*)

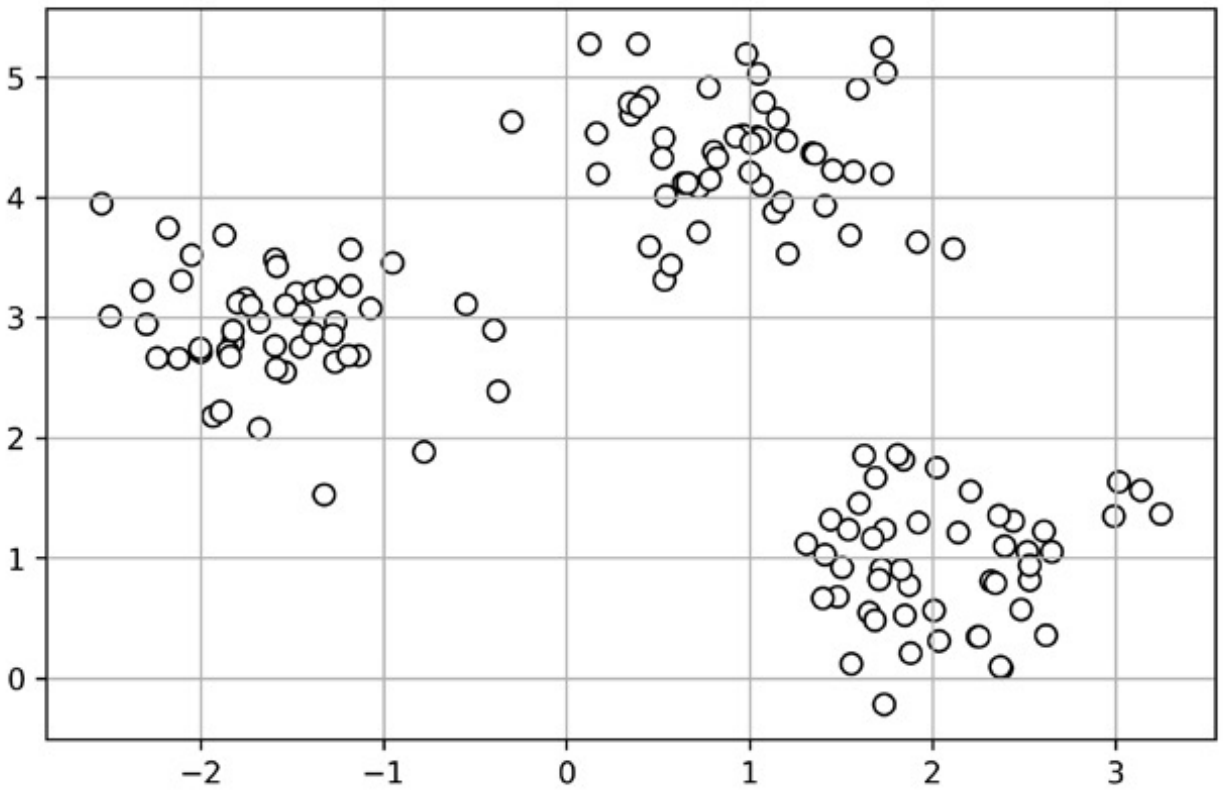
```
from sklearn.datasets import make_blobs
X, y = make_blobs(n_samples=150,
      ... n_features=2,
      ... centers=3,
      ... cluster_std=0.5,
      ... shuffle=True,
      ... random_state=0)

import matplotlib.pyplot as plt
plt.scatter(X[:,0],
      ... X[:,1],
      ... c='white’,
      ... marker='o’,
      ... edgecolor='black’,
      ... s=50)
plt.grid()
plt.show()
```



- k-means algorithm that can be summarized by 4 steps:

1. *Randomly* pick *k* centroids from the sample points as initial cluster centers.

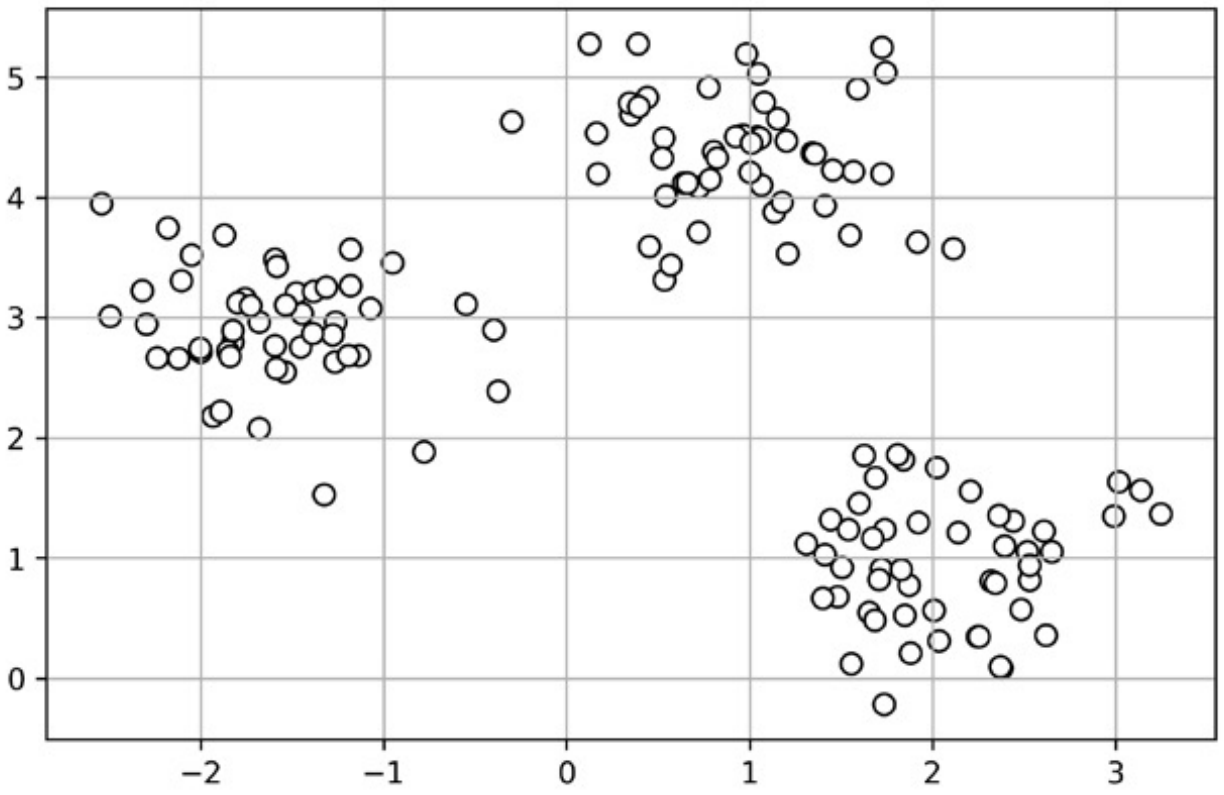
2. Assign each sample to the *nearest centroid* $\mu^{(j)}$, $j \in \{1,\ldots,k\}$.



3. Move the centroids to the center of the samples that were assigned to it.

4. Repeat 2 and 3 until:

- the cluster assignments do not change
- user-defined tolerance
- maximum number of iterations is reached



- Similarities between objects

- Distance for clustering samples with continuous features is the *squared Euclidean* distance between two points *x* and *y* in *m-dimensional* space

$$d(\boldsymbol{x}, \boldsymbol{y})^2 = \sum_{j=1}^{m} \left(x_j - y_j\right)^2 = \|\boldsymbol{x} - \boldsymbol{y}\|_2^2$$

- Index *j* refers to the jth dimension (feature column) of the sample points *x* and *y*

$$d(\boldsymbol{x}, \boldsymbol{y})^2 = \sum_{j=1}^{m} \left(x_j - y_j \right)^2 = \left\| \boldsymbol{x} - \boldsymbol{y} \right\|_2^2$$

- The k-means algorithm as a simple optimization problem → an iterative approach for minimizing the within-cluster **Sum of Squared Errors (SSE)**

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

- $\boldsymbol{\mu}^{(j)}$ is the representative point (centroid) for cluster *j*
- w(i, j) =1 if the sample $\boldsymbol{x}^{(i)}$ is in cluster *j*
- w(i, j) = 0 otherwise

$$SSE = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

- A problem with k-means is that one or more clusters can be empty
- **If a cluster is empty** → the algorithm will search for the sample that is farthest away from the centroid of the empty cluster.
- Then, it will reassign the centroid to be this farthest point

K-Means Code

```
from sklearn.cluster import Kmeans

km = KMeans(n_clusters=3,
    ... init='random’,
    ... n_init=10,
    ... max_iter=300,
    ... tol=1e-04,
    ... random_state=0)

y_km = km.fit_predict(X)
```

```
km = KMeans(n_clusters=3,
```

- number of desired clusters to 3

```
... n_init=10,
```

- run the k-means clustering algorithms <u>10 times independently</u> with different random centroids to choose the final model as the one with the lowest SSE.

```
... max_iter=300,
```

- the maximum number of iterations for each single run

```
... tol=1e-04,
```

- Tolerance limit of the SSE calculation

```
... n_init=10,
```

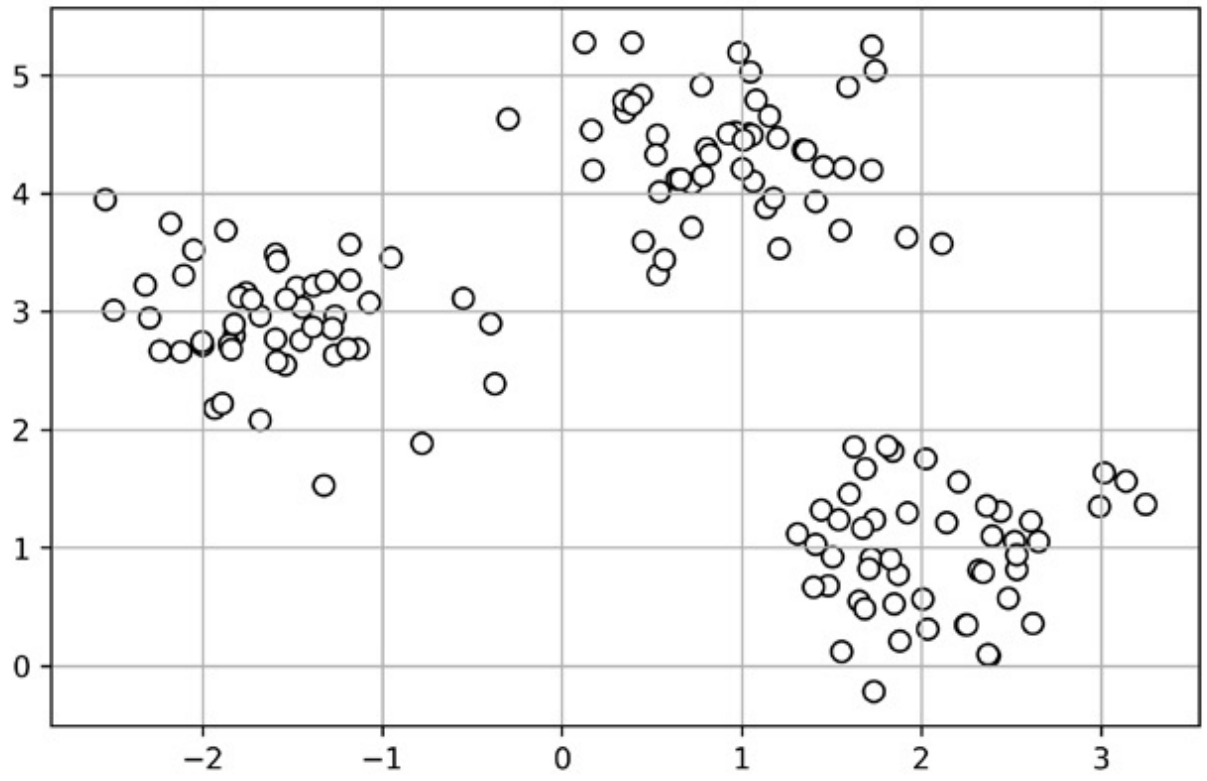
- run the k-means clustering algorithms <u>10 times independently</u> with different random centroids to choose the final model as the one with the lowest SSE.

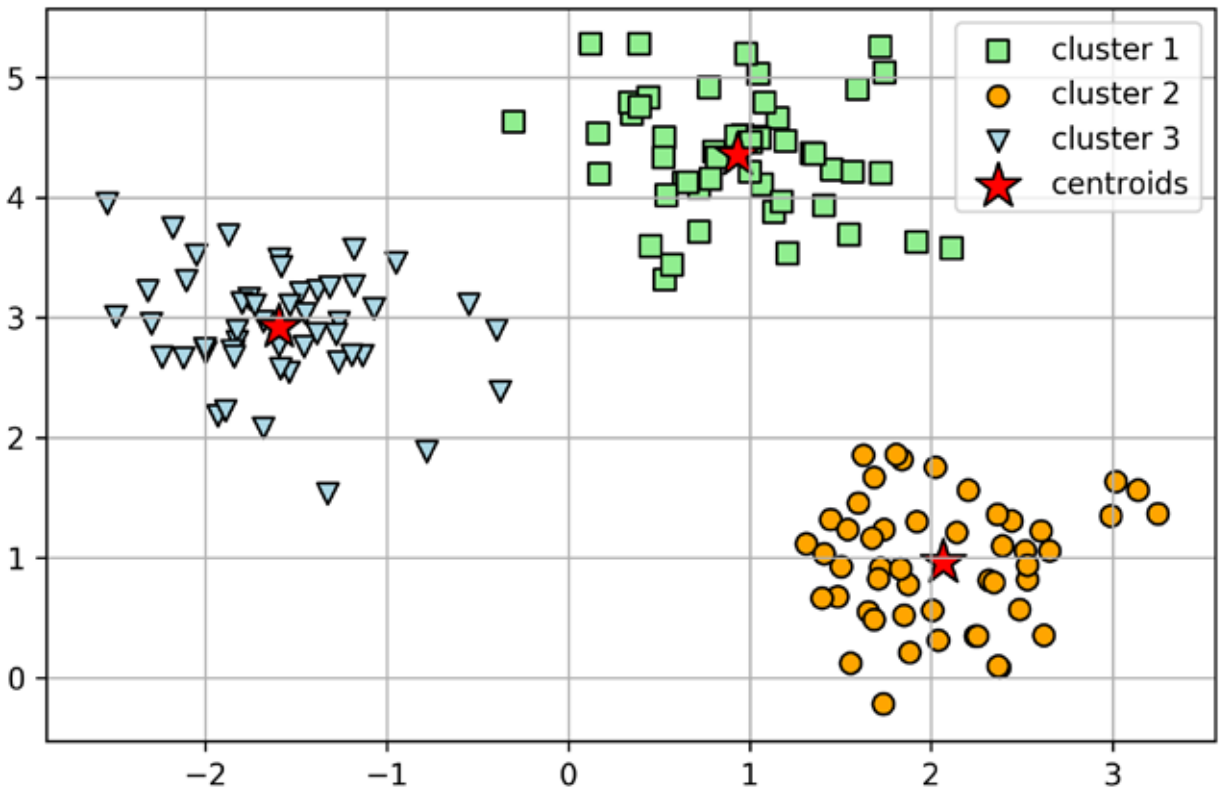
```
... max_iter=300,
```

- the maximum number of iterations for each single run

```
plt.scatter(X[y_km == 0, 0],
     ... X[y_km == 0, 1],
     ... s=50, c='lightgreen’,
     ... marker='s', edgecolor='black’,
     ... label='cluster 1’)

plt.scatter(X[y_km == 1, 0],
     ... X[y_km == 1, 1],
     ... s=50, c='orange’,
     ... marker='o', edgecolor='black’,
     ... label='cluster 2')
```

```
plt.scatter(X[y_km == 2, 0],
     ... X[y_km == 2, 1],
     ... s=50, c='lightblue’,
     ... marker='v', edgecolor='black’,
     ... label='cluster 3’)

plt.scatter(km.cluster_centers_[:, 0],
     ... km.cluster_centers_[:, 1],
     ... s=250, marker='*’,
     ... c='red', edgecolor='black’,
     ... label='centroids’)

plt.legend(scatterpoints=1)
plt.grid()
plt.show()
```


5
4
3
2
1
0
−2
−1
0
1
2
3


cluster 1
cluster 2
cluster 3
centroids
5
4
3
2
1
0
−2
−1
0
1
2
3

- Another drawback of k-means → must specify the number of clusters
  - What the hell!!!!?

- **Is this a good method for Engineering projects?**
  - **Why or why not?**

K-Means++ Clustering

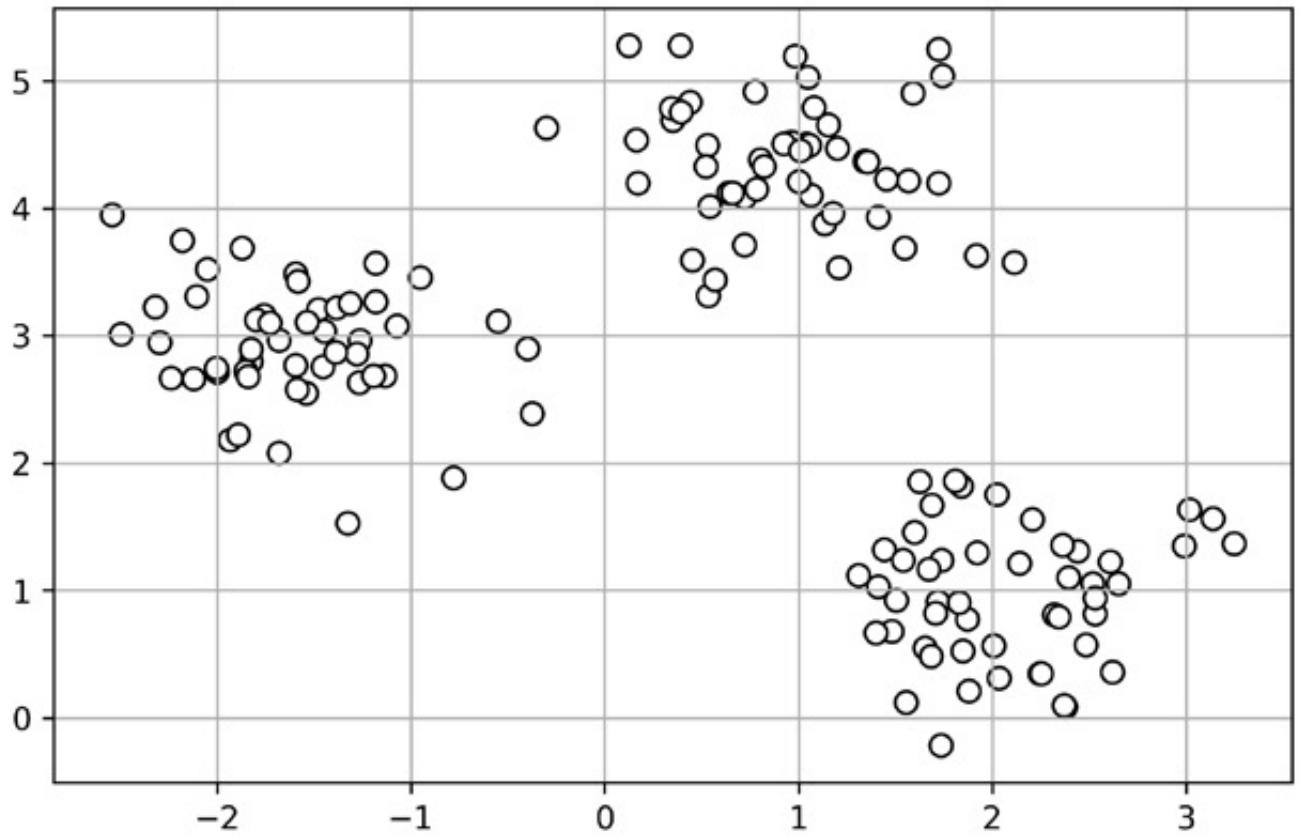
- **k-means++** → strategy is to place the initial centroids far away from each other

- To use k-means++ with scikit-learn's KMeans object → just need to set the parameter to 'k-means++'.

- k-means++ algorithm that can be summarized by 6 steps:

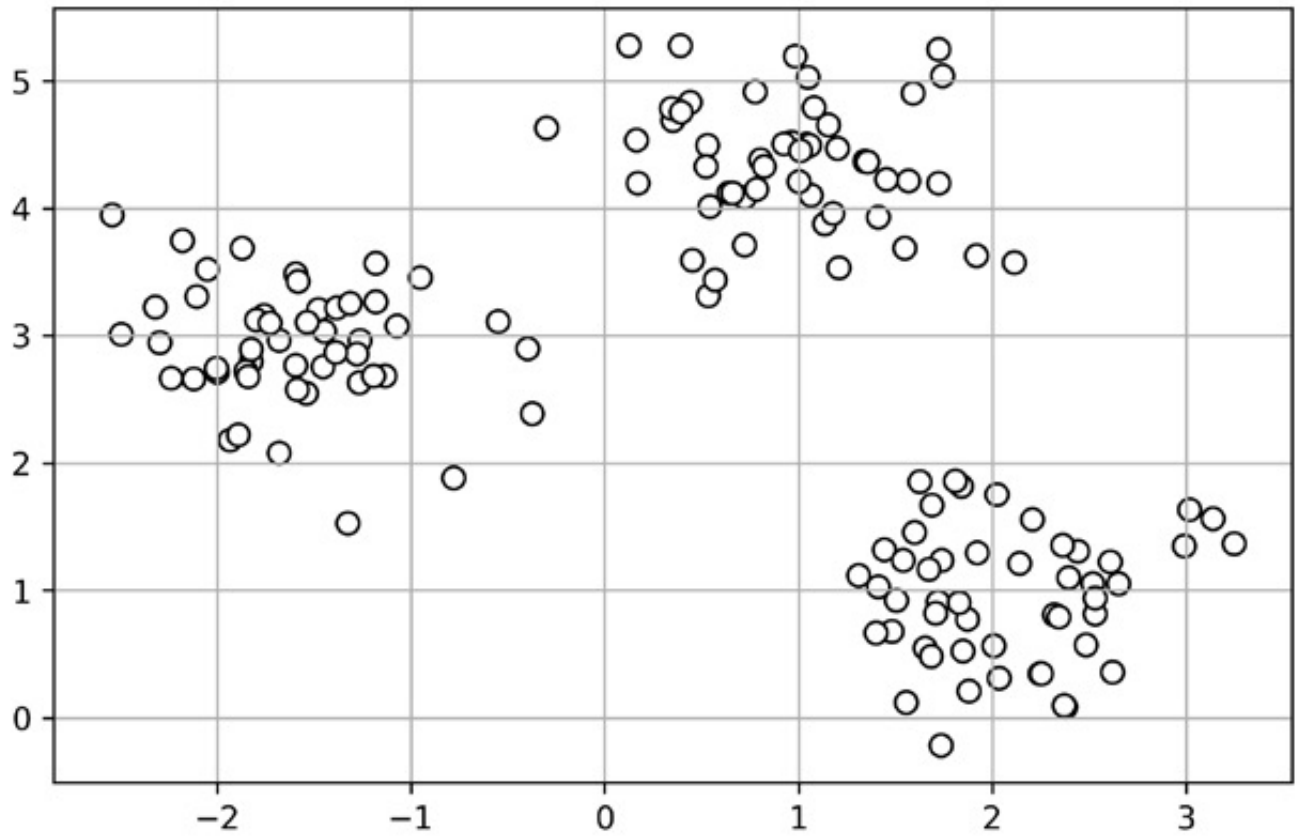
1. Initialize an empty set ***M*** to store the *k* centroids being selected.

2. Randomly choose the first centroid $\boldsymbol{\mu}^{(j)}$ from the input samples and assign it to ***M***.



- k-means++ algorithm that can be summarized by 6 steps:

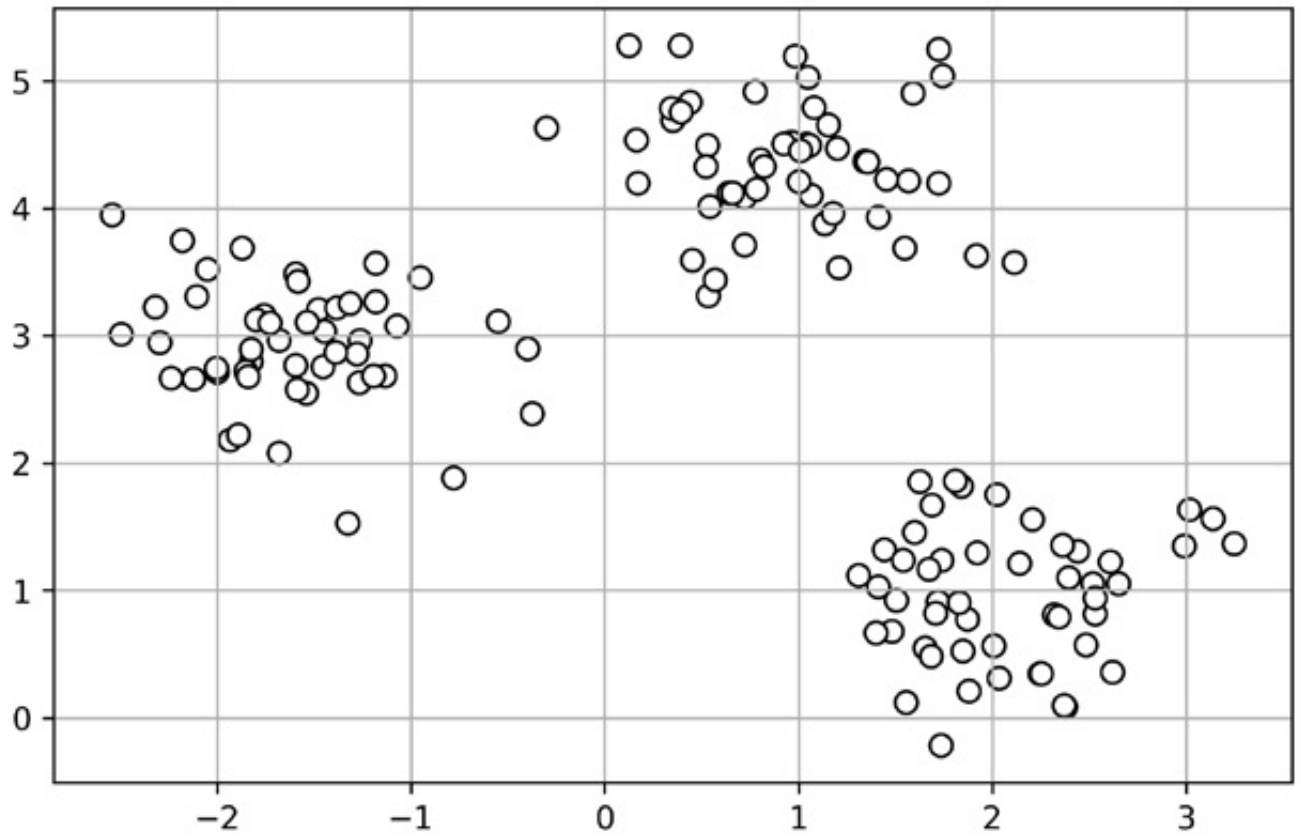
3. For each sample $\boldsymbol{x}^{(i)}$ that is not in **M**, find the minimum squared distance $d(\boldsymbol{x}^{(i)}, \mathbf{M})^2$ to any of the centroids in **M**.



- k-means++ algorithm that can be summarized by 6 steps:

4. Randomly select the next centroid $\boldsymbol{\mu}^{(p)}$, use a weighted probability distribution equal to

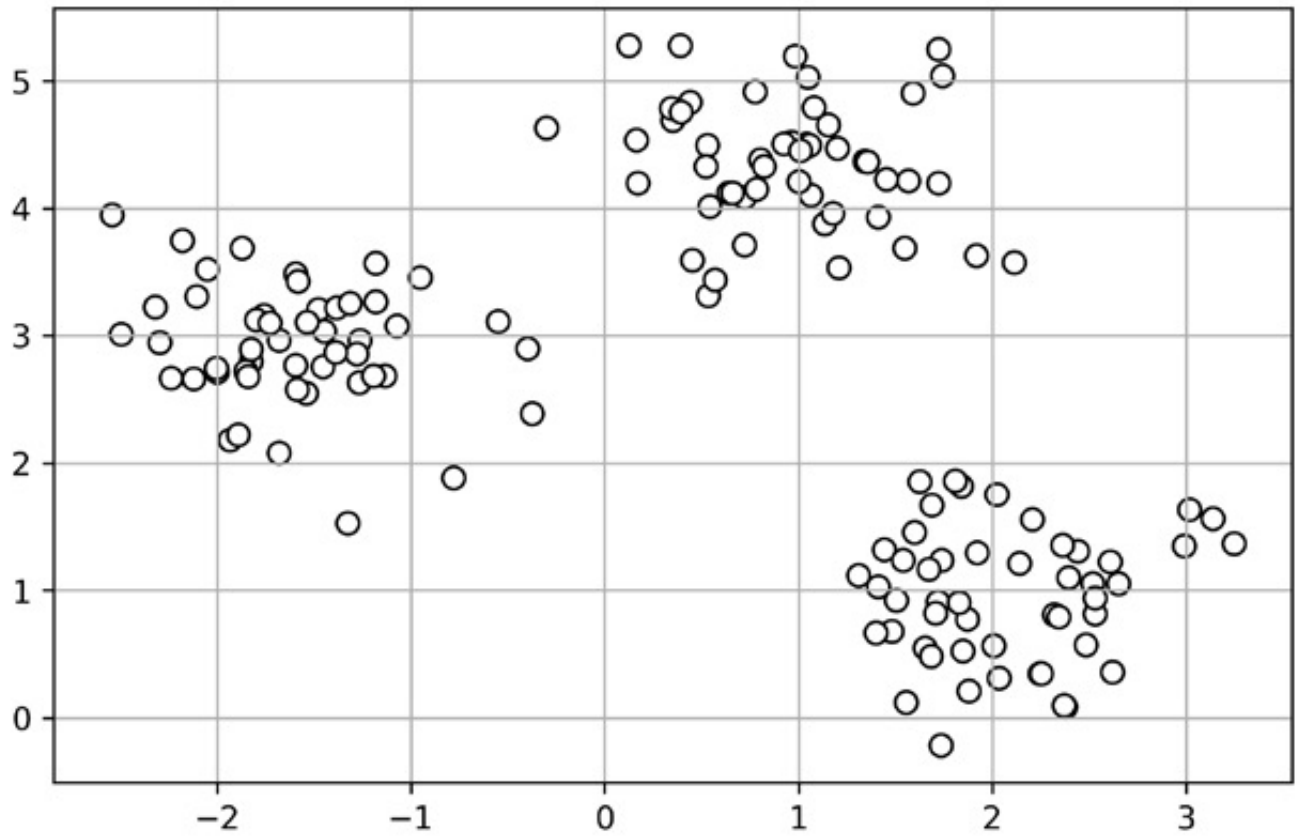
$$\frac{d(\boldsymbol{\mu}^{(p)}, \mathbf{M})^2}{\sum_i d(\boldsymbol{\mu}^{(p)}, \mathbf{M})^2}$$



- k-means++ algorithm that can be summarized by 6 steps:

5. Repeat steps 2 and 3 until k centroids are chosen.

6. Proceed with the classic k-means algorithm.



Hard vs. Soft Clustering

- **Hard clustering** → where each sample in a dataset is assigned to exactly one cluster, as in the k-means algorithm

- **Soft clustering** (aka - fuzzy clustering) assign a sample to one or more clusters

- A popular example of soft clustering is the **fuzzy C-means (FCM) algorithm**

- The **FCM** procedure is very similar to k-means.

- However → replace the *hard cluster assignment* <u>with probabilities</u> for each point belonging to each cluster.

- In k-means → the cluster membership of a sample *x* with a sparse vector of binary values:

$$\begin{bmatrix} \mu^{(1)} \rightarrow 0 \\ \mu^{(2)} \rightarrow 1 \\ \mu^{(3)} \rightarrow 0 \end{bmatrix}$$

- The index position with value 1 indicates the cluster centroid $\boldsymbol{\mu}^{(j)}$ the sample is assigned to
  - assuming $k = 3$
  - $j \in \{1, 2, 3\}$

- In contrast, a membership vector in FCM:

$$\begin{bmatrix} \boldsymbol{\mu}^{(1)} \rightarrow 0.10 \\ \boldsymbol{\mu}^{(2)} \rightarrow 0.85 \\ \boldsymbol{\mu}^{(3)} \rightarrow 0.05 \end{bmatrix}$$

Sum should equal to one

- The FCM algorithm in 4 key steps:

1. Specify the number of *k* centroids and randomly assign the cluster memberships for each point

2. Compute the cluster centroids $\boldsymbol{\mu}^{(j)}$, $j \in \{1,\ldots, k\}$

3. Update the cluster memberships for each point.

4. Repeat steps 2 and 3 **<u>until</u>** the membership coefficients do not change, **<u>or</u>** a user-defined tolerance or maximum number of iterations is reached.

- Objective function of FCM

- Looks very similar to the within cluster sum-squared-error in k-means

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{m\,(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{m\,(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

- The membership indicator $w^{(i,\,j)}$ is not a binary value as in k-means ($w^{(i,\,j)} \in \{0,1\}$)

- But a real value that denotes the cluster membership probability ($w^{(i,\,j)} \in [0,1]$)).

$$J_m = \sum_{i=1}^{n} \sum_{j=1}^{k} w^{m\,(i,j)} \left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2^2$$

- The exponent *m:* any number greater than or equal to one (typically *m*=2), is the so-called fuzziness coefficient (or simply **fuzzifier**)
  - Controls <u>the degree of fuzziness</u>

- The larger the value of *m* → the smaller the cluster membership $w^{(i,\,j)}$ becomes → leads to fuzzier clusters.

- The cluster membership probability itself is calculated

$$w^{(i,j)} = \left[\sum_{p=1}^{k} \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(p)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

- Back to the three cluster centers as in the previous k-means example

- The calculation of the membership of the $\boldsymbol{x}^{(i)}$ sample belonging to the $\boldsymbol{\mu}^{(j)}$ cluster:

$$w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

$$w^{(i,j)} = \left[\left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(1)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(2)} \right\|_2} \right)^{\frac{2}{m-1}} + \left(\frac{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|_2}{\left\| \boldsymbol{x}^{(i)} - \boldsymbol{\mu}^{(3)} \right\|_2} \right)^{\frac{2}{m-1}} \right]^{-1}$$

- $\boldsymbol{\mu}^{(j)}$ of a cluster itself is calculated as the mean of all samples weighted by the degree to which each sample belongs to that cluster ($w^{(i,j)}$)

$$\boldsymbol{\mu}^{(j)} = \frac{\sum_{i=1}^{n} w^{m(i,j)} \boldsymbol{x}^{(i)}}{\sum_{i=1}^{n} w^{m(i,j)}}$$

- FCM Cons:
  - Each iteration in FCM is more expensive than an iteration in k-means
  - The FCM algorithm is currently not implemented in scikit-learn

- FCM Pros:
  - FCM typically requires fewer iterations overall to reach convergence
  - Both k-means and FCM produce very similar clustering outputs

Elbow Method

- To quantify the quality of clustering → use intrinsic metrics—such as the within-cluster SSE (distortion) to compare the performance of different k-means clusterings

- Don't need to compute the within-cluster SSE explicitly when using scikit-learn → as it is already accessible via the `inertia_` attribute after fitting a KMeans model:

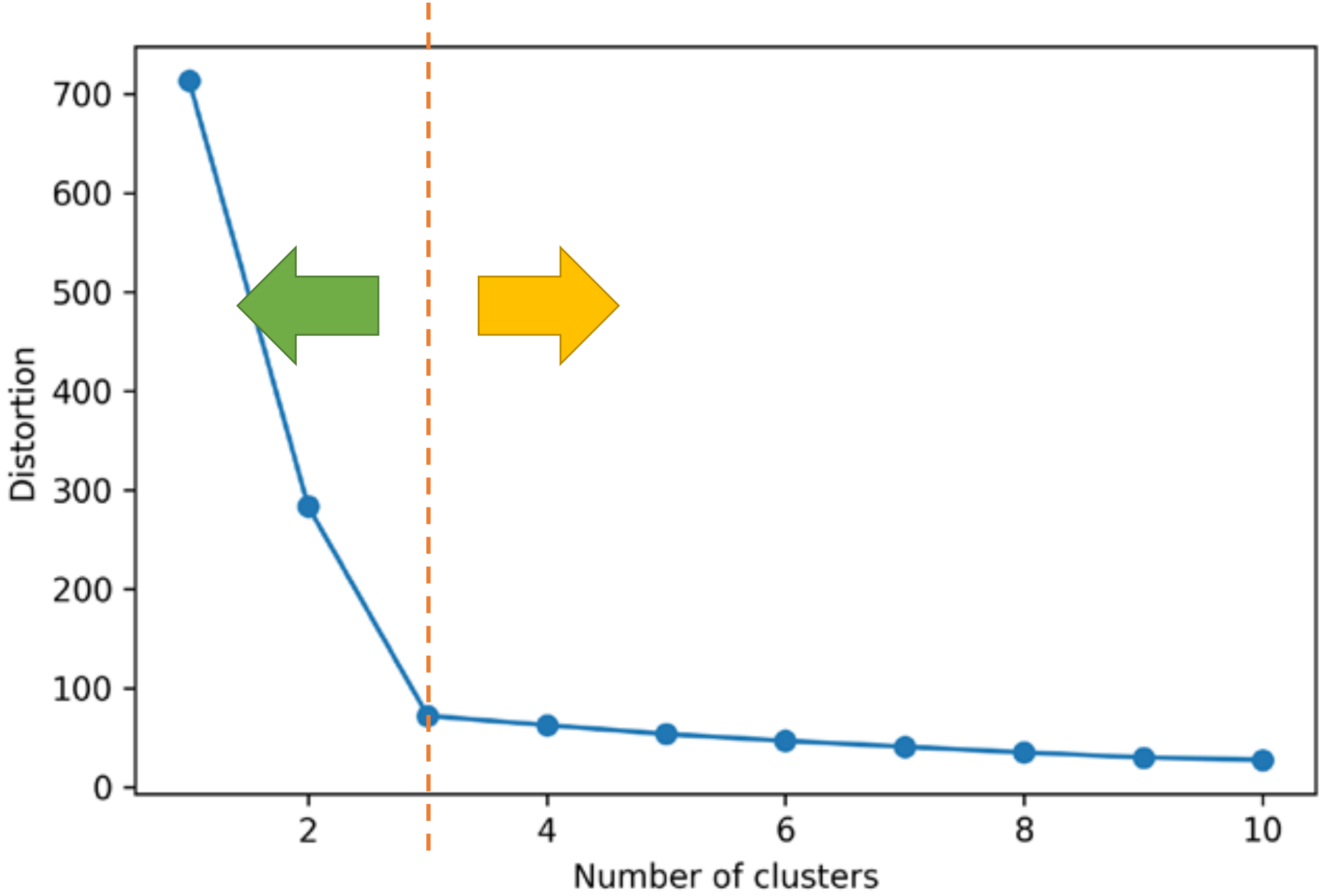
```
print('Distortion: %.2f' % km.inertia_)
Distortion: 72.48
```

- Elbow method → estimates the optimal number of clusters *k* for a given task

- If *k* increases → the distortion will decrease.
  - Because the samples will be closer to the centroids they are assigned to.

- The idea behind the elbow method is to identify the value of *k* where the distortion begins to increase most rapidly

```
distortions = []
for i in range(1, 11):
    ... km = KMeans(n_clusters=i,
    ... init='k-means++',
    ... n_init=10,
    ... max_iter=300,
    ... random_state=0)
km.fit(X)
distortions.append(km.inertia_)

plt.plot(range(1,11), distortions, marker='o')
plt.xlabel('Number of clusters')
plt.ylabel('Distortion')
plt.show()
```


700
600
500
400
300
200
100
0
Distortion
2
4
6
8
10
Number of clusters

Silhouette Plots

- **Silhouette analysis** can be used *as a graphical tool* to plot a measure of how tightly grouped the samples in the clusters are.

- To calculate the silhouette coefficient of a single sample in a dataset is done in 3 steps.

1. Calculate the **cluster cohesion** $a^{(i)}$ as the average distance between a sample $\boldsymbol{x}^{(i)}$ and all other points in the same cluster.

2. Calculate the **cluster separation** $b^{(i)}$ from the next closest cluster as the average distance between the sample $\boldsymbol{x}^{(i)}$ and all samples in the nearest cluster.

3. Calculate the **silhouette $s^{(i)}$** as the difference between $a^{(i)}$ and $b^{(i)}$ divided by the greater of the two

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\left\{b^{(i)}, a^{(i)}\right\}}$$

- The silhouette coefficient is bounded in the range -1 to 1

- An ideal silhouette coefficient of 1 **if** $b^{(i)}$ >> $a^{(i)}$
  - Since $b^{(i)}$ quantifies how <u>dissimilar</u> a sample is to other clusters
  - $a^{(i)}$ → how <u>similar</u> it is to the other samples in its own cluster

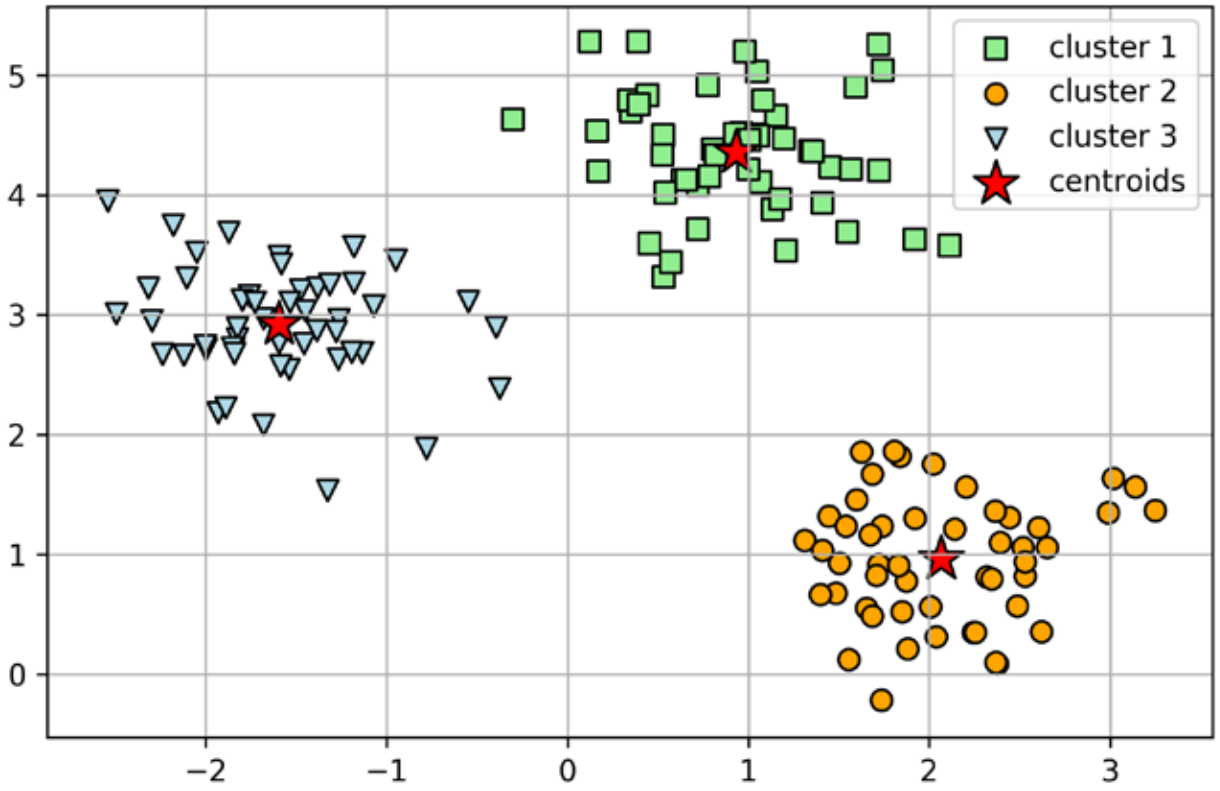
- `silhouette_samples` from scikit-learn's metric module, and optionally, the `silhouette_scores` function can be imported for convenience.

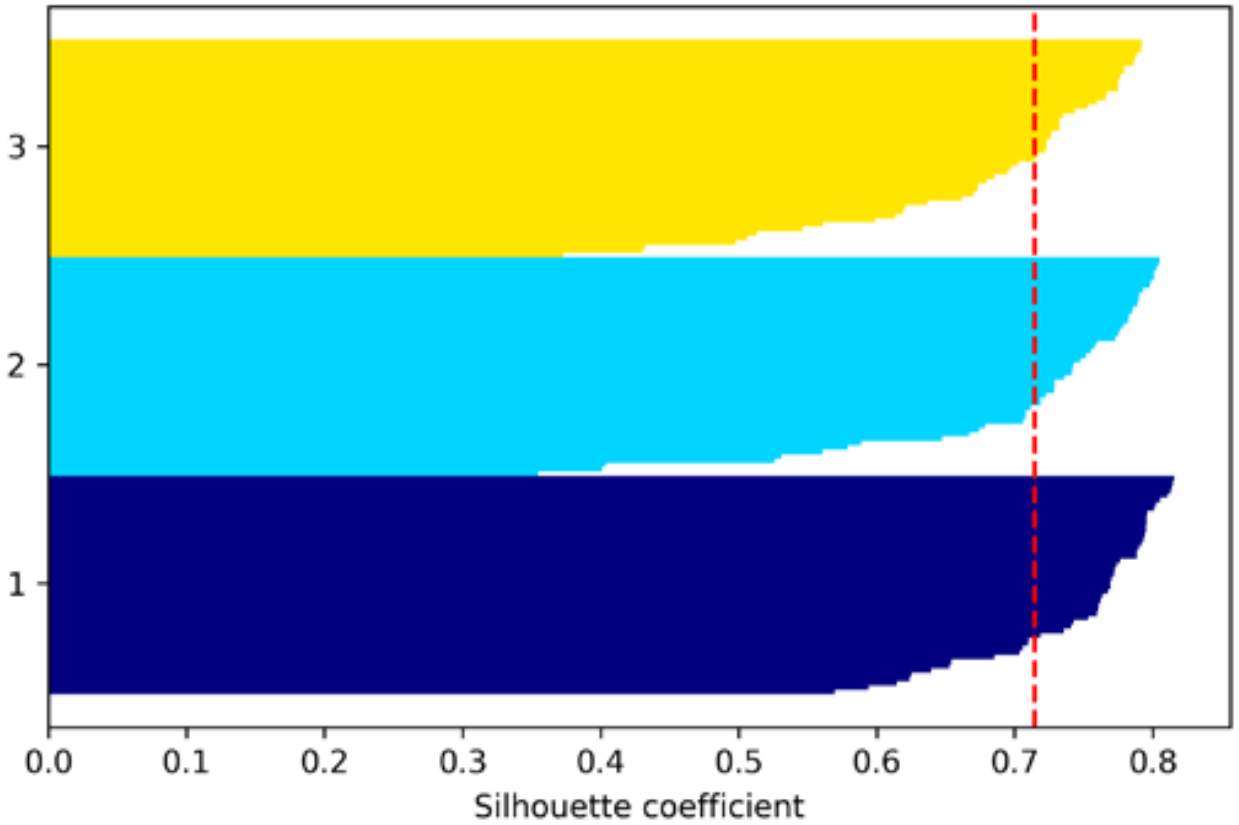
- The `silhouette_scores` function calculates the average silhouette coefficient across all samples, which is equivalent to:

```
numpy.mean(silhouette_samples(…))
```

```
km = KMeans(n_clusters=3,
        ... init='k-means++', n_init=10,
        ... max_iter=300, tol=1e-04, random_state=0)
y_km = km.fit_predict(X)

import numpy as np
from matplotlib import cm
from sklearn.metrics import silhouette_samples

cluster_labels = np.unique(y_km)
n_clusters = cluster_labels.shape[0]
silhouette_vals = silhouette_samples(X,
                                     ... y_km,
                                     ... metric='euclidean')
```

$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\left\{b^{(i)}, a^{(i)}\right\}}$$





$$s^{(i)} = \frac{b^{(i)} - a^{(i)}}{\max\left\{b^{(i)}, a^{(i)}\right\}}$$