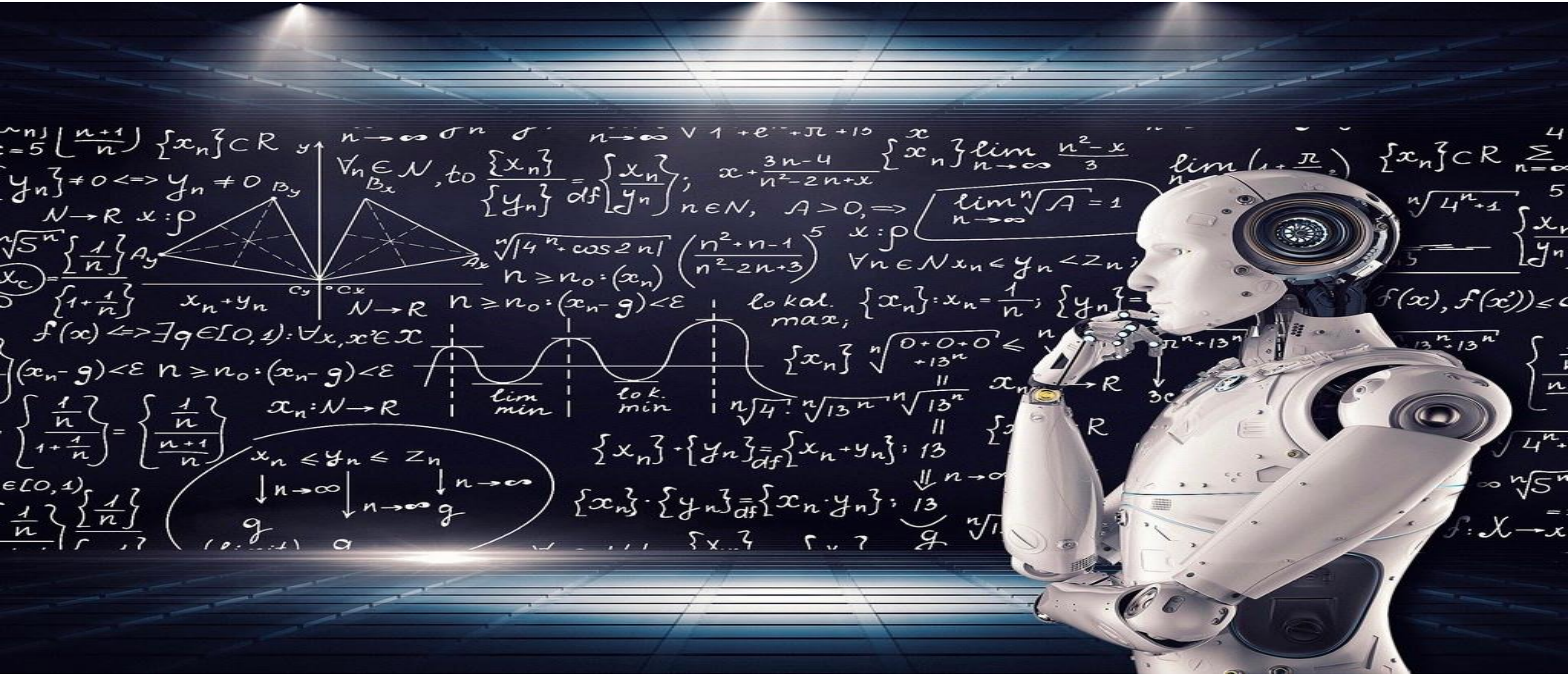
Unsupervised Learning



IDC HERZLIYA

The goal

- We don’t know the labels of the data
- What can we know about data with no labels?
    - Structure
    - Regularity
    - Grouping
- This means we can find the similarity between the instances
- This is called Unsupervised Learning

Unsupervised Learning

- The task of inferring a function to describe hidden structure from unlabeled data
- This doesn't mean that there is no "guidance"
- There are several usages for this learning task

Clustering

- Partition unlabeled examples into subsets of clusters
  - Patterns within a cluster are very similar
  - Patterns in different clusters are very different
- A cluster is a volume of high-density points separated from other clusters by a relatively low-density volumes
- Clustering is all about modeling the underlying structure of the feature space
  - As opposed to supervised learning which aims at minimizing the loss with respect to a specified target (e.g. label, value)

Clustering

- A clustering algorithm will always find clusters, even if there are none …
- There is no “best” clustering method
  - But we can tell how good is the solution it provides
- The various approaches may vary in the following
  - Data Representation
  - Similarity Measures
  - Hypotheses Space
  - Objective Function
  - Evaluation criteria for quality and validity of the solution

Similarity

- How to decide if samples are similar or not?
- We first need to know how to measure similarity
- Distance – the closer the instances are, the more similar they are
    - Non-negativity:
        - $d(x_1, x_2) \geq 0$
    - Identity of indiscernible:
        - $d(x_1, x_2) = 0 \Leftrightarrow x_1 = x_2$
    - Symmetry:
        - $d(x_1, x_2) = d(x_2, x_1)$
    - Triangle inequality:
        - $d(x_1, x_2) \leq d(x_1, x_3) + d(x_3, x_2)$
- We already saw some different distance methods (Manhattan, Euclidean, Infinity Norm)

Similarity

- How to use the similarity to create clusters?
- Simple approach:
  - Start from randomly selected un-clustered instance and insert it to a new cluster
  - Insert to the cluster all instances that have distance (the similarity measure) lower then some threshold to one of the instances in the cluster – repeat till no instance was added to the cluster
  - Repeat the 2 steps till all the instances were clustered

Objective function

- We want to achieve ‘good’ clustering
- How can we know if the partition of the instances to some k clusters is good or not?
  - Maximize between classes
  - Minimize within classes

K-Means

- Centroid-based clustering, clusters are represented by a central vector, which may not necessarily be a member of the data set
- Given a set of observations ($\mathbf{x}_1$, …, $\mathbf{x}_n$), where each $\mathbf{x}_i$ is a *d*-dimensional vector
- Centroid-based algorithms aim to partition the *n* observations into *k* (≤ *n*) clusters $\{C_1, \ldots, C_k\}$ so as to minimize the within-cluster sum of squares (WCSS):

$$\sum_{i=1}^{k} \sum_{x \in C_i} \|x - C_i\|^2$$

where c_i is the cluster center

Find the best solution

- How many possibility for partition n instances into k clusters?
- There are approximately $\frac{k^n}{k!}$ ways to partition the n elements into k clusters
- And if we also searching for k

$$\sim \sum_{k=1}^{n} \frac{k^n}{k!}$$

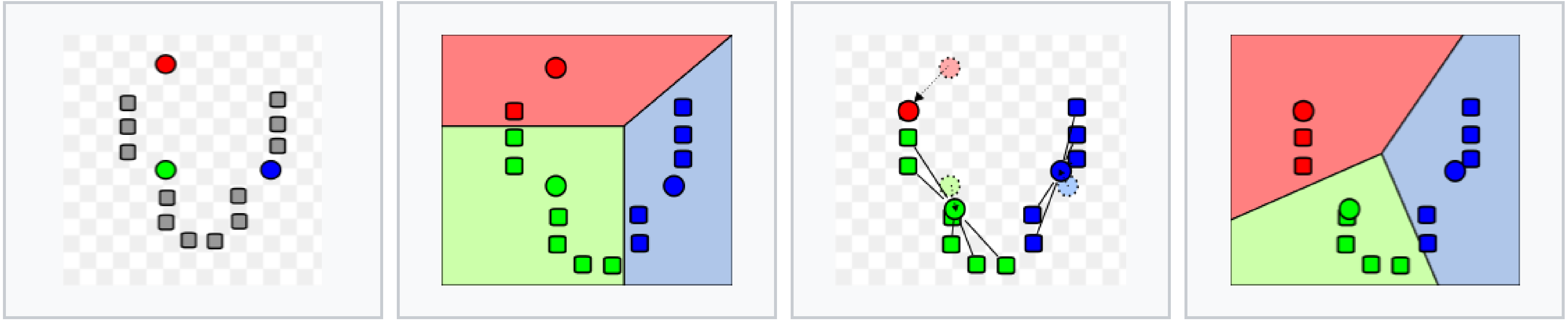
- Finding the optimal solution to this optimization problem is NP-Hard even for k = 2 Thus, a variety of heuristic algorithms are generally used

K-Means

- Initialize randomly the k-means $\mu_1, \cdots, \mu_k$
- Repeat
    - For each instance
        - Assign it to the nearest cluster w.r.t its mean μ_i
    - Re-computes μ_i for each cluster
- Until no change in $\mu_1, \cdots, \mu_k$ (or any other stopping condition)
- Return $\mu_1, \cdots, \mu_k$

* Usually uses simple Euclidean distance in feature space

K-Means Demonstration



1. *k*=3 initial "means" are randomly generated within the data domain

2. *k* clusters are created by associating every observation with the nearest mean

3. The centroid of each of the *k* clusters becomes the new mean

4. Steps 2 and 3 are repeated until convergence has been reached

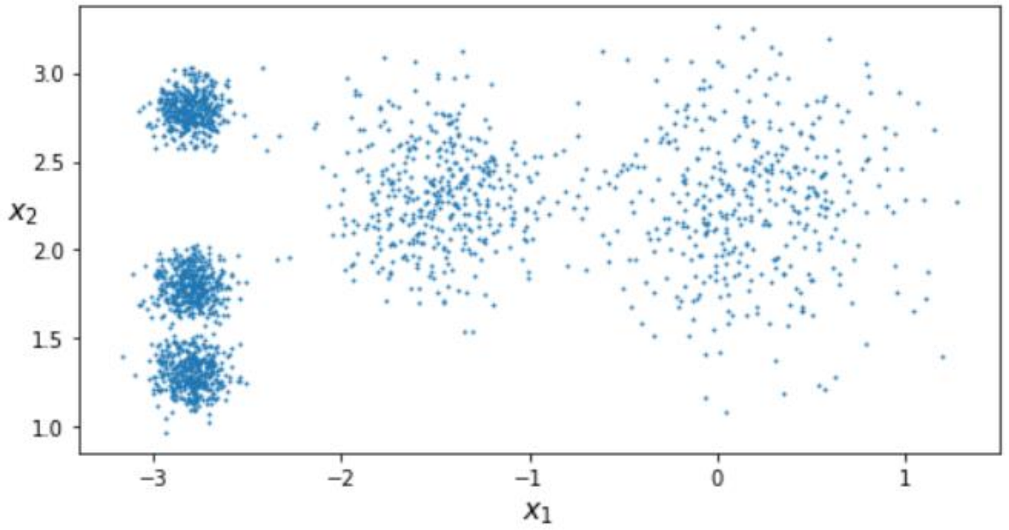
K-Means Clustering

- Let's start by generating some blobs:

```python
from sklearn.datasets import make_blobs

blob_centers = np.array(
    [[ 0.2, 2.3],
     [-1.5, 2.3],
     [-2.8, 1.8],
     [-2.8, 2.8],
     [-2.8, 1.3]])
blob_std = np.array([0.4, 0.3, 0.1, 0.1, 0.1])

X, y = make_blobs(n_samples=2000, centers=blob_centers,
                  cluster_std=blob_std)
```

```python
def plot_clusters(X, y=None):
    plt.scatter(X[:, 0], X[:, 1], c=y, s=1)
    plt.xlabel("$x_1$", fontsize=14)
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

```python
plt.figure(figsize=(8, 4))
plot_clusters(X)
plt.show()
```



K-Means Clustering

- Let's train a K-Means on this dataset
- It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans

k = 5
kmeans = KMeans(n_clusters=k, random_state=0)
y_pred = kmeans.fit_predict(X)
y_pred
```

```
array([1, 4, 4, ..., 1, 1, 3])
```

- K-Means preserves the labels of the instances it was trained on
- The label of an instance is the index of the cluster that instance gets assigned to:

```
kmeans.labels_
```

```
array([1, 4, 4, ..., 1, 1, 3])
```

```
y_pred is kmeans.labels_
```

```
True
```

K-Means Clustering

- The **cluster_centroids_** variable holds the 5 centroids (cluster centers):

```
kmeans.cluster_centers_
```

```
array([[-2.80379109,  1.30017997],
       [ 0.17864202,  2.27506362],
       [-2.80180006,  2.79464334],
       [-1.46332098,  2.31083369],
       [-2.80446936,  1.79959228]])
```

- Of course, we can predict the labels of new instances:

```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)
```

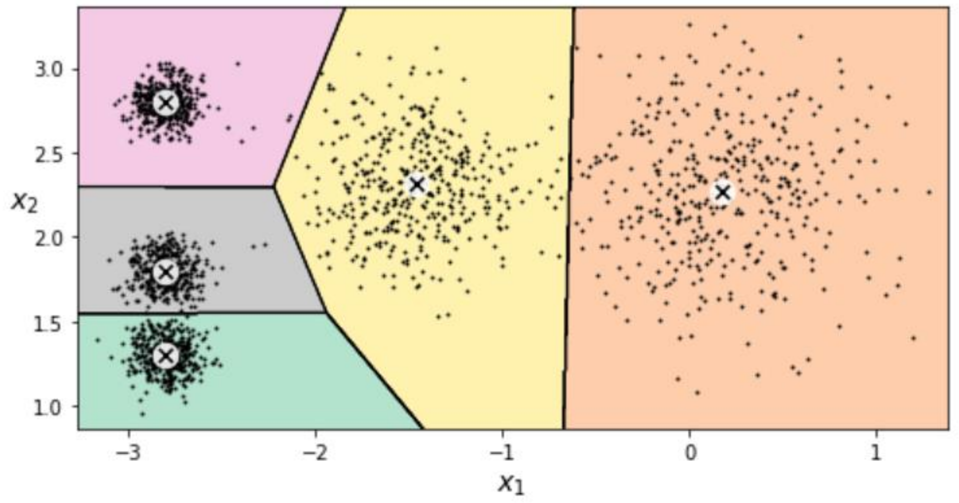
```
array([1, 1, 2, 2])
```

Decision Boundaries

- Let's plot the model's decision boundaries. This gives us a *Voronoi diagram*:

```python
def plot_decision_boundaries(clusterer, X, resolution=1000,
                             show_centroids=True):
    mins = X.min(axis=0) - 0.1
    maxs = X.max(axis=0) + 0.1
    xx, yy = np.meshgrid(np.linspace(mins[0], maxs[0], resolution),
                         np.linspace(mins[1], maxs[1], resolution))
    Z = clusterer.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                cmap='Pastel2')
    plt.contour(Z, extent=(mins[0], maxs[0], mins[1], maxs[1]),
                linewidths=1, colors='k')
    plot_data(X)
    if show_centroids:
        plot_centroids(clusterer.cluster_centers_)
    plt.xlabel("$x_1$", fontsize=14)
    plt.ylabel("$x_2$", fontsize=14, rotation=0)
```

```python
plt.figure(figsize=(8, 4))
plot_decision_boundaries(kmeans, X)
plt.show()
```



K-Means Clustering

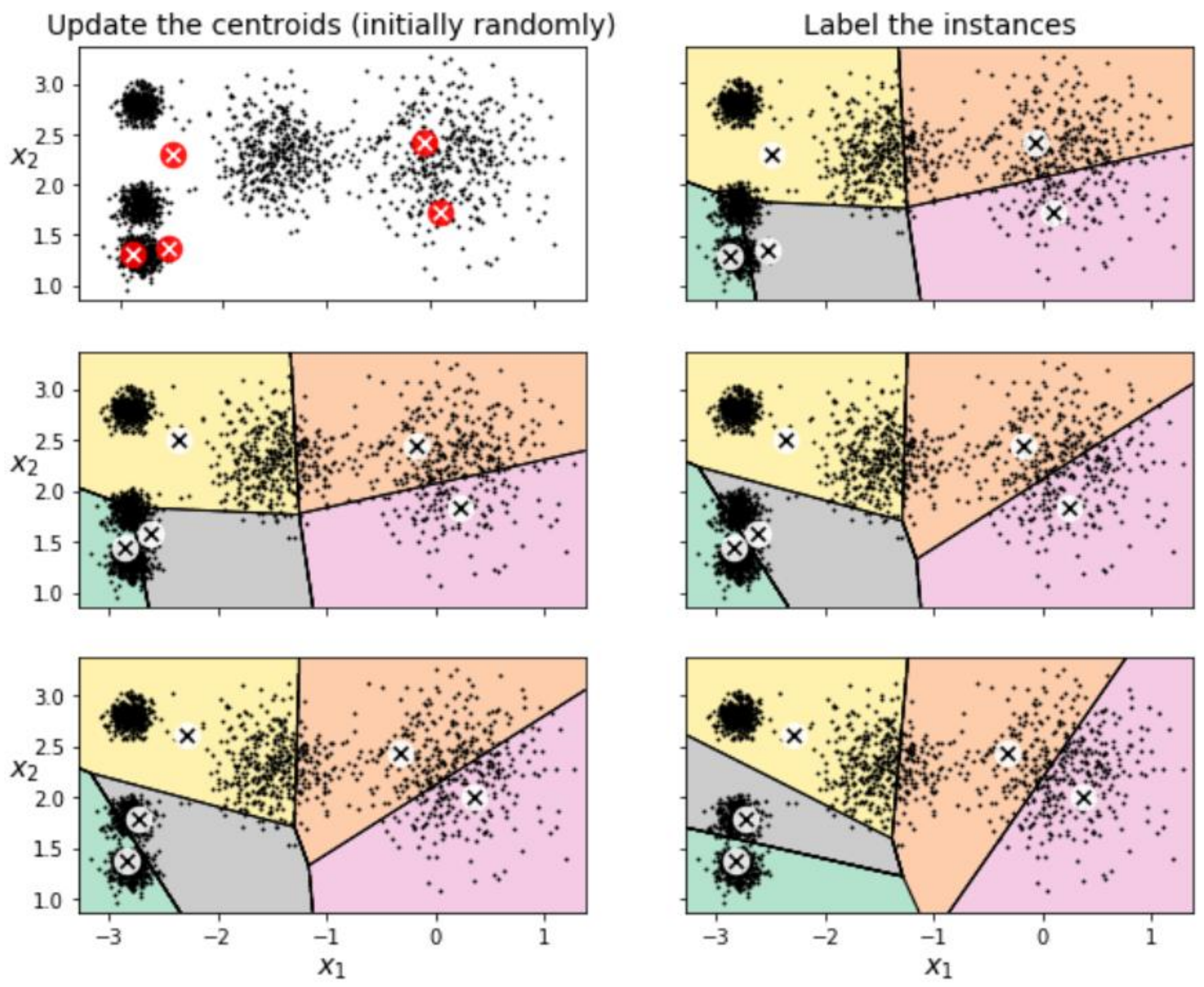
- Let's run the K-Means algorithm for 1, 2 and 3 iterations, to see how the centroids move around:

```
kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=1, random_state=0)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=2, random_state=0)
kmeans_iter3 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=3, random_state=0)
kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)
```

```
KMeans(algorithm='full', copy_x=True, init='random', max_iter=300,
    n_clusters=5, n_init=1, n_jobs=None, precompute_distances='auto',
    random_state=0, tol=0.0001, verbose=0)
```

- The K-Means class applies an optimized algorithm by default
- To get the original K-Means algorithm, you need to set init="random", n_init=1 and algorithm="full"

K-Means Clustering



K-Means

- Is this algorithm find the solution for the optimization problem?

$$minimize \sum_{i=1}^{k} \sum_{x \in D_i} \|x - \mu_i\|^2$$

- We can rewrite the function:

$$minimize \frac{1}{m} \sum_{i=1}^{m} \left\|x_i - \mu_{c_i}\right\|^2$$

Where μ_{c_i} is the mean of the cluster that x_i belong to

K-Means

- We can show that at each iteration the function $\frac{1}{m}\sum_{i=1}^{m}\left\|x_i - \mu_{c_i}\right\|^2$ is reduced, why?
  - The assignment of x_i to a cluster:
    - If x_i doesn't change its cluster – stay the same
    - If x_i does change its cluster, it assign to the nearest cluster – reduce
  - Re-compute means:
    - The average will give the min square error – reduce

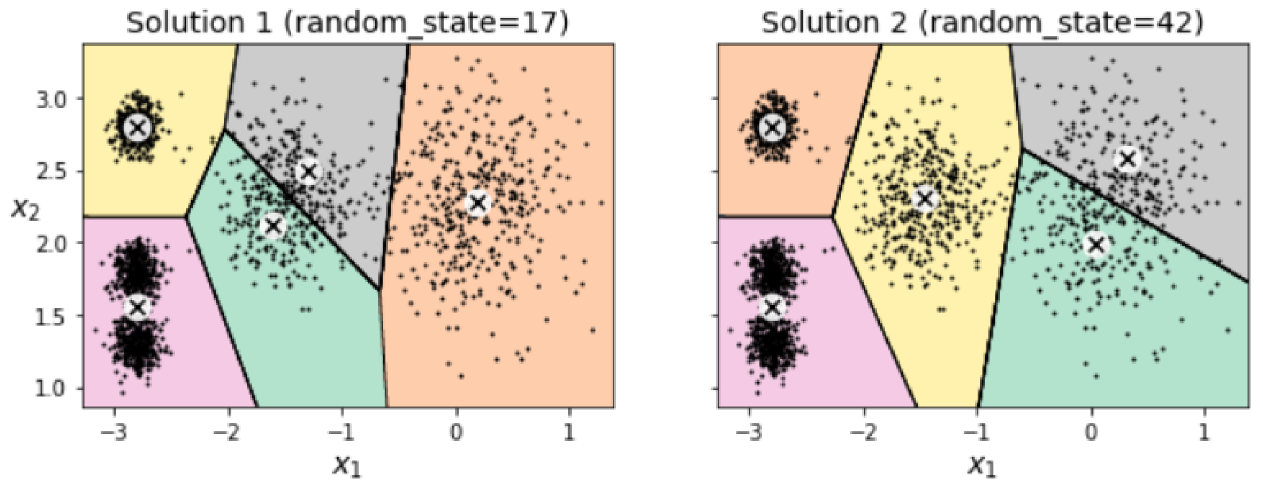
K-Means Variability

- In the original K-Means algorithm, the centroids are just initialized randomly, and the algorithm simply runs a single iteration to gradually improve the centroids
- A problem with this approach is that if you run K-Means multiple times (or with different random seeds), it can converge to very different solutions (local minima problem):

```
def plot_clusterer_comparison(clusterer1, clusterer2, X,
                              title1=None, title2=None):
    clusterer1.fit(X)
    clusterer2.fit(X)

    plt.figure(figsize=(10, 3.2))

    plt.subplot(121)
    plot_decision_boundaries(clusterer1, X)
    if title1:
        plt.title(title1, fontsize=14)

    plt.subplot(122)
    plot_decision_boundaries(clusterer2, X, show_ylabels=False)
    if title2:
        plt.title(title2, fontsize=14)
```

K-Means Variability

```
kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=17)
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1,
                          algorithm="full", random_state=42)

plot_clusterer_comparison(kmeans_rnd_init1, kmeans_rnd_init2, X,
                          "Solution 1 (random_state=17)",
                          "Solution 2 (random_state=42)")
```



Inertia

- To select the best model, we will need a way to evaluate a K-Mean model's performance
- Unfortunately, clustering is an unsupervised task, so we do not have the targets
- But at least we can measure the distance between each instance and its centroid
- This is the idea behind the *inertia* metric:

```
kmeans.inertia_
```

```
210.90540191468097
```

- As you can easily verify, inertia is the sum of the squared distances between each training instance and its closest centroid:

```
X_dist = kmeans.transform(X)
np.sum(X_dist[np.arange(len(X_dist)), kmeans.labels_]**2)
```

```
210.90540191468136
```

- The score() method returns the negative inertia
- That’s s because a predictor's score() method respects the "*great is better*" rule

```
kmeans.score(X)
```

```
-210.90540191468136
```

Multiple Initializations

- One approach to solve the variability issue is to simply run the K-Means algorithm multiple times with different random initializations, and select the solution that minimizes the inertia
- For example, here are the inertias of the two "bad" models shown in the previous figure:

```
kmeans_rnd_init1.inertia_
```

```
238.48246617500146
```

```
kmeans_rnd_init2.inertia_
```

```
219.0489599322056
```

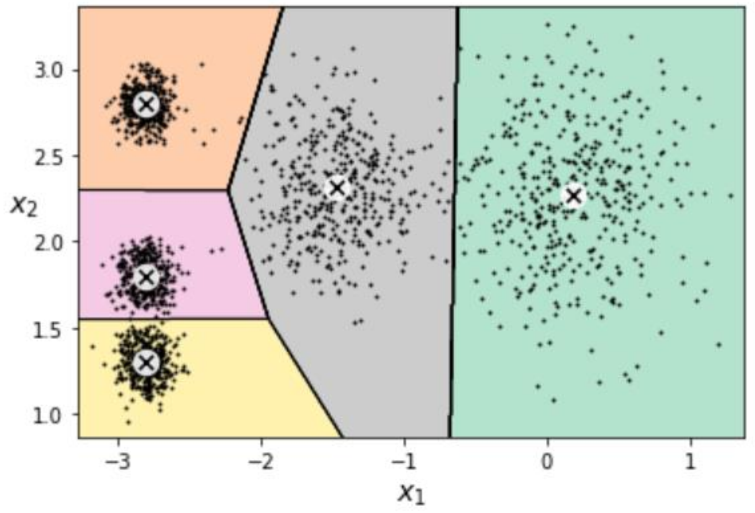
- As you can see, they have a higher inertia than the first "good" model we trained, which means they are probably worse

Multiple Initializations

- When you set the n_init hyperparameter, Scikit-Learn runs the original algorithm n_init times, and selects the solution that minimizes the inertia
- By default, Scikit-Learn sets n_init=10

```
kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random", n_init=10,
                             algorithm="full", random_state=17)
kmeans_rnd_10_inits.fit(X)
print(kmeans_rnd_10_inits.inertia_)

plot_decision_boundaries(kmeans_rnd_10_inits, X)
plt.show()
```

```
210.9054019146808
```

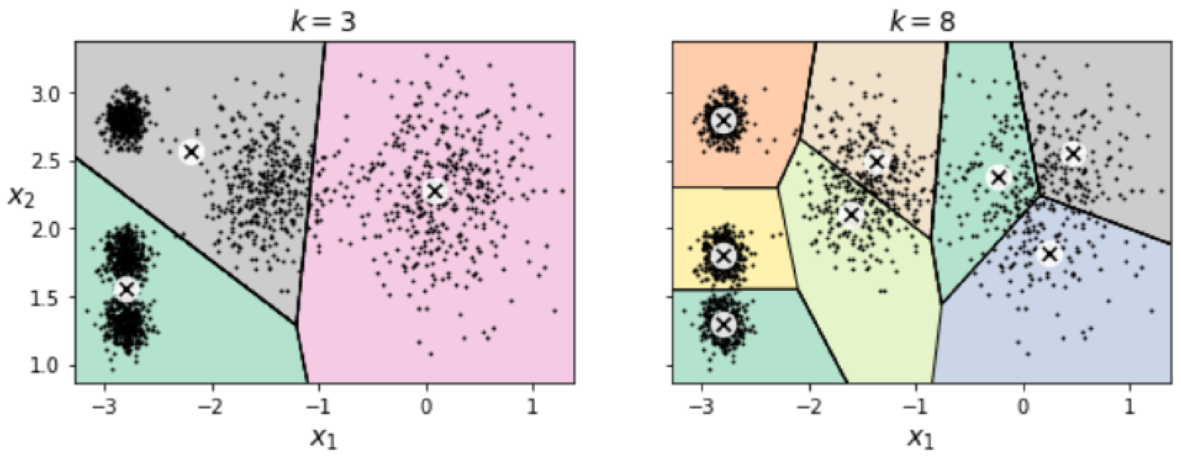


- As you can see, we end up with the initial model, which is certainly the optimal K-Means solution (at least in terms of inertia)

Finding the Optimal Number of Clusters

- What if the number of clusters was set to a lower or greater value than 5?

```
kmeans_k3 = KMeans(n_clusters=3, random_state=0)
kmeans_k8 = KMeans(n_clusters=8, random_state=0)

plot_clusterer_comparison(kmeans_k3, kmeans_k8, X, "$k=3$", "$k=8$")
```



- These two models don't look great. What about their inertias?

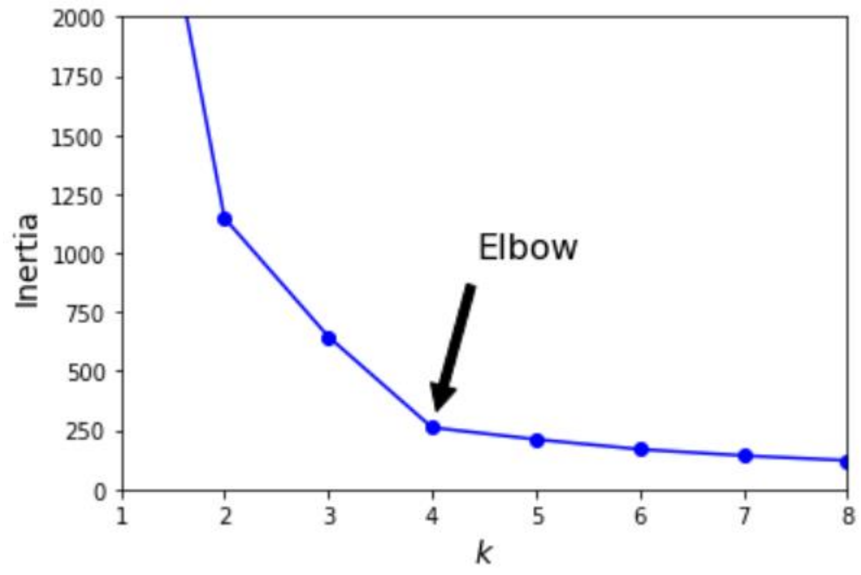
```
print("k3 inertia:", kmeans_k3.inertia_)
print("k8 inertia:", kmeans_k8.inertia_)
```

```
k3 inertia: 646.2252909343712
k8 inertia: 121.60198303182476
```

Finding the Optimal Number of Clusters

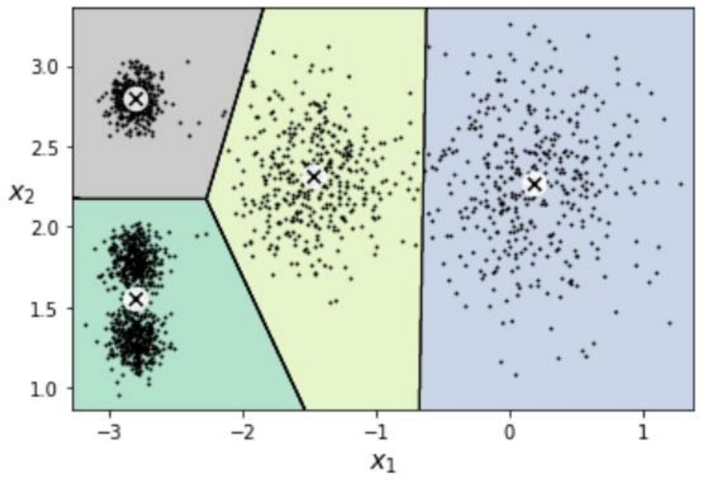
- We cannot simply take the value of *k* that minimizes the inertia, since it keeps getting lower as we increase *k*
- Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be
- However, we can plot the inertia as a function of *k* and analyze the resulting curve:

```
plt.plot(range(1, 10), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.annotate('Elbow',
             xy=(4, inertias[3]),
             xytext=(0.55, 0.55),
             textcoords='figure fraction',
             fontsize=16,
             arrowprops=dict(facecolor='black', shrink=0.1)
            )
plt.axis([1, 8, 0, 2000])
plt.show()
```



Finding the Optimal Number of Clusters

- As you can see, there is an elbow at *k* = 4, which means that less clusters than that would be bad, and more clusters would not help much and might cut clusters in half
- So k = 4 is a pretty good choice
- In this example it is not perfect since it means that the two blobs in the lower left will be considered as just a single cluster, but it's a pretty good clustering nonetheless

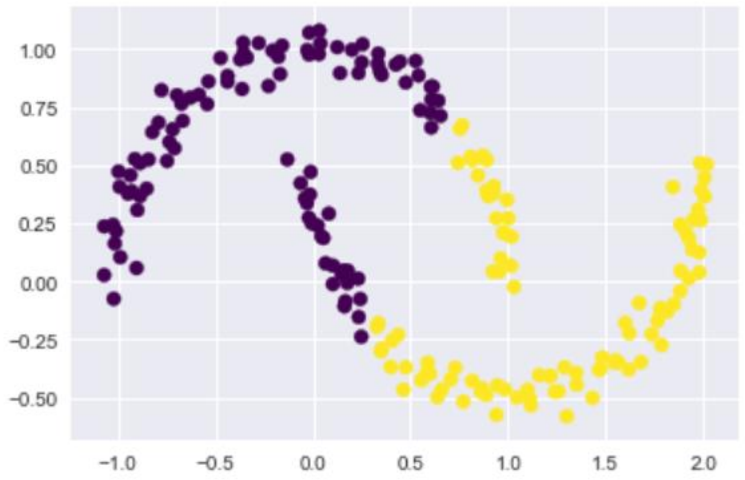


Hard Clustering vs. Soft Clustering

- Hard – assumes that each instance is given a “hard” assignment to exactly one cluster
- Soft – gives probabilities that an instance belongs to each one of the clusters
- How can we use it in k-means?
  - With distance
  - For example:
    - For each instance calculate a distance vector from all means
    - Normalized the vector
    - Use weighted average to calculate the new means

Another problem…

- Another caveat of k-means is that its assumption that points will be closer to their own cluster center than to others means that the boundaries between *k*-means clusters will always be linear
- Thus, the *k*-means method is not suitable for discovering clusters with nonconvex shapes or clusters of very different size
- Moreover, it is sensitive to noise and outlier data points because a small number of such data can substantially influence the mean value



Hierarchical Clustering

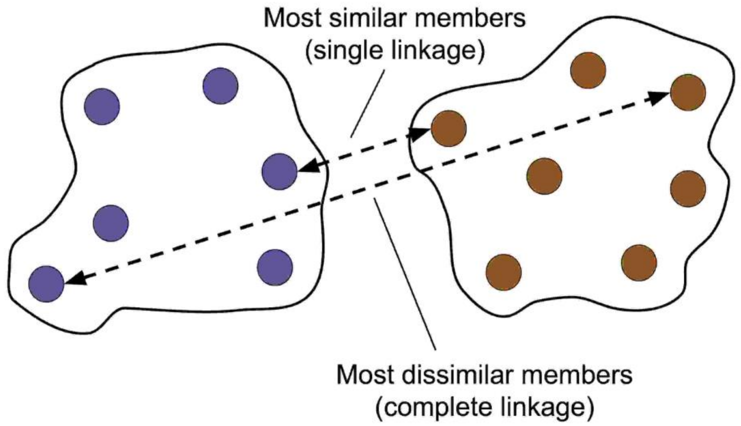
- The two main approaches to hierarchical clustering are **agglomerative and divisive** hierarchical clustering
- In agglomerative clustering, we start with each sample as an individual cluster and merge the closest pairs of clusters until only one cluster remains
- In divisive hierarchical clustering, we start with one cluster that encompasses all our samples, and we iteratively split the cluster into smaller clusters until each cluster only contains one sample.

Agglomerative Clustering

- Agglomerative hierarchical clustering is an iterative procedure that can be summarized by the following steps:
  - Compute the distance matrix of all samples
  - Represent each data point as a singleton cluster
  - Merge the two closest clusters based on the chosen linkage criterion
  - Update the similarity matrix
  - Repeat steps 2-4 until one single cluster remains

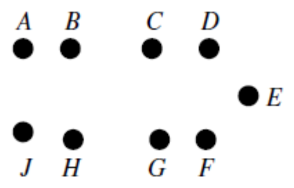
Linkage Measures

- The linkage criterion determines which distance to use between sets of observation
- The clustering algorithm will merge the pairs of clusters that minimize this criterion
- Common linkage measures:
  - **Single linkage** uses the minimum of the distances between all observations of the two sets
  - **Complete linkage** uses the maximum distances between all observations of the two sets
  - **Ward** minimizes the variance of the clusters being merged
  - **Average** uses the average of the distances of each observation of the two sets

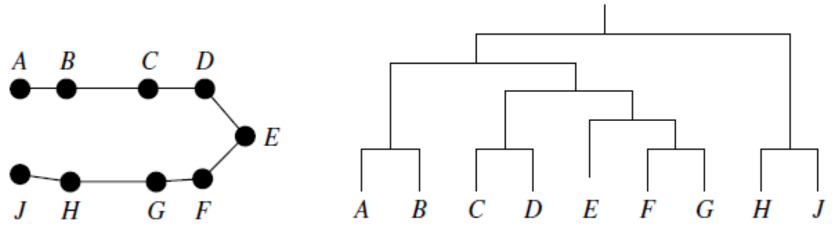


Dendrogram

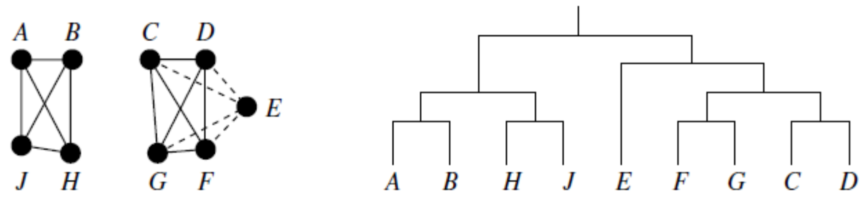
- The main output of Hierarchical Clustering is a *dendrogram,* which shows the hierarchical relationship between the clusters:



(a) Data set



(b) Clustering using single linkage



(c) Clustering using complete linkage

Example

- You have the following points in your dataset:
  {(1,2), (4,8), (3,9), (7,3), (4,3), (2,4), (5,2), (3,5), (2,5), (6,6)}
- We’ll run an agglomerative hierarchical clustering on this dataset
- Use a Manhattan distance metric and the single linkage criterion

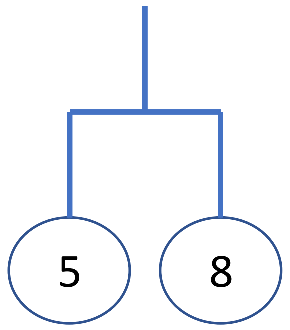
Example

Point	X1	X2
0	1	2
1	4	8
2	3	9
3	7	3
4	4	3
5	2	4
6	5	2
7	3	5
8	2	5
9	6	6

Point	Dist 0	Dist 1	Dist 2	Dist 3	Dist 4	Dist 5	Dist 6	Dist 7	Dist 8	Dist 9
0										
1	9									
2	9	2								
3	7	8	10							
4	4	5	7	3						
5	3	6	6	6	3					
6	4	7	9	3	2	5				
7	5	4	4	6	3	2	5			
8	4	5	5	7	4	1	6	1		
9	9	4	6	4	5	6	5	4	5	

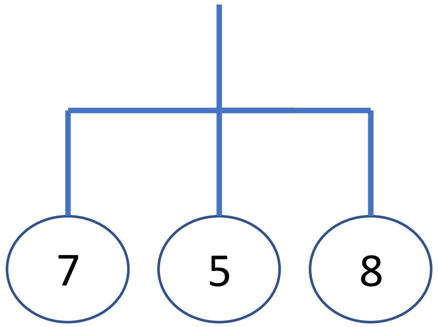
Example

	Dist 0	Dist 1	Dist 2	Dist 3	Dist 4	Dist 5,8	Dist 6	Dist 7	Dist 9
0									
1	9								
2	9	2							
3	7	8	10						
4	4	5	7	3					
5,8	3	5	5	6	3				
6	4	7	9	3	2	5			
7	5	4	4	6	3	1	5		
9	9	4	6	6	5	5	5	4	

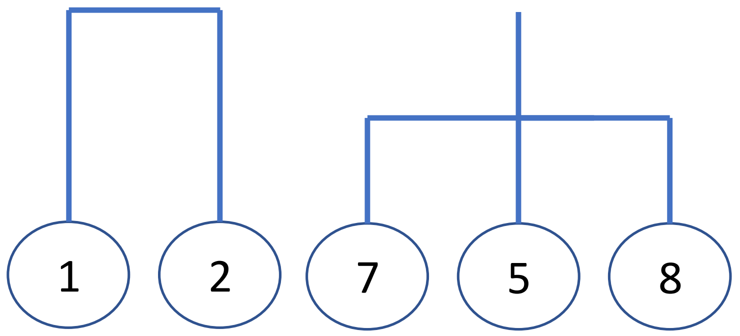


Example

	Dist 0	Dist 1	Dist 2	Dist 3	Dist 4	Dist 7,5,8	Dist 6	Dist 7
0								
1	9							
2	9	$\textcircled{2}$						
3	7	8	10					
4	4	5	7	3				
7,5,8	3	4	4	6	3			
6	4	7	9	3	$\textcircled{2}$	5		
9	9	4	6	6	5	4	5	

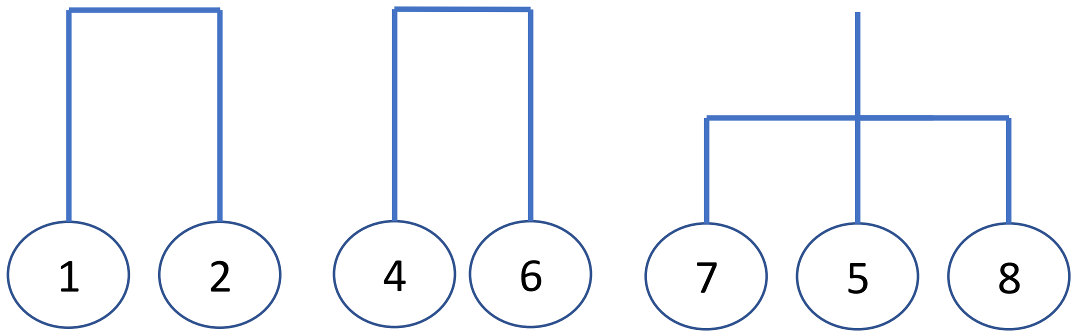


Example



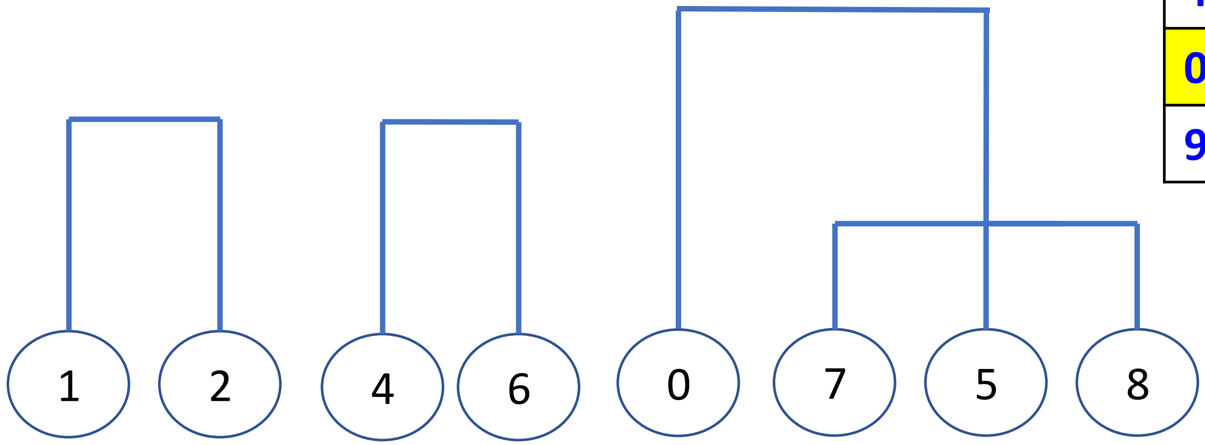
	0	1,2	3	4	7,5,8	6	9
0							
1,2	9						
3	7	8					
4	4	5	3				
7,5,8	3	4	6	3			
6	4	7	3	2	5		
9	9	4	6	5	4	5	

Example



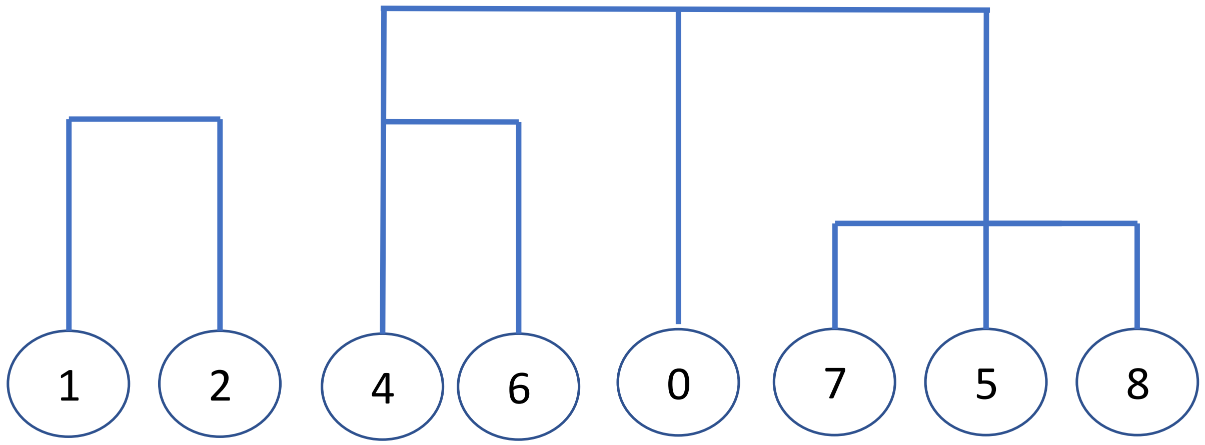
	0	1,2	3	4,6	7,5,8	9
0						
1,2	9					
3	7	8				
4,6	4	5	3			
7,5,8	3	4	6	3		
9	9	4	6	5	4	

Example



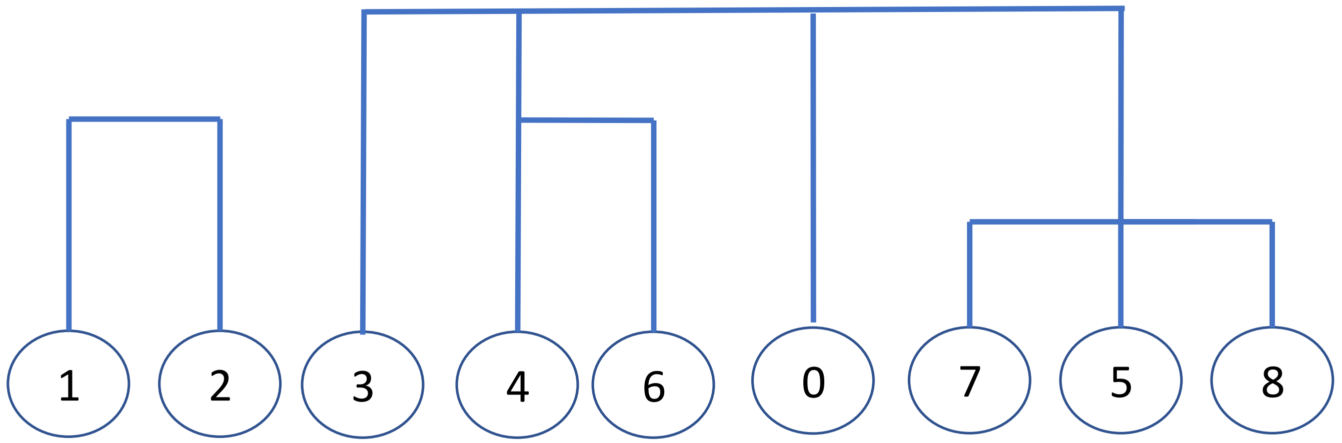
	1,2	3	4,6	0,7,5,8	9
1,2					
3	8				
4,6	5	3			
0,7,5,8	4	6	3		
9	4	6	5	4	

Example



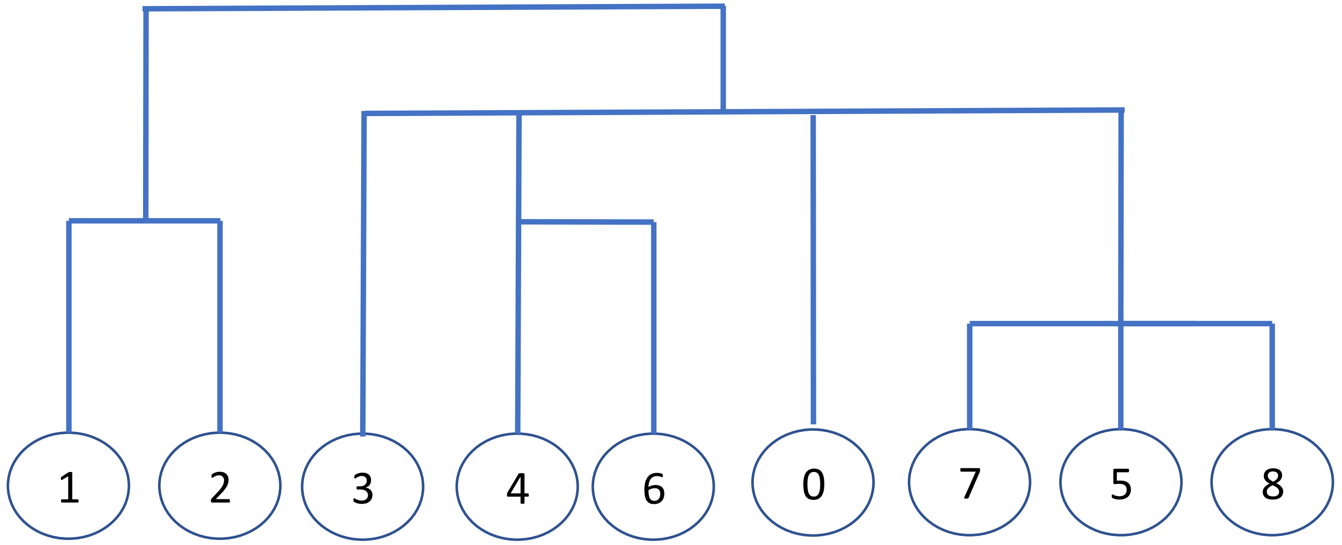
	1,2	3	4,6,0,7, 5,8	9
1,2				
3	8			
4,6,0,7, 5,8	4	3		
9	4	6	4	

Example



	1,2	3,4,6,0, 7,5,8	9
1,2			
3,4,6,0, 7,5,8	4		
9	4	4	

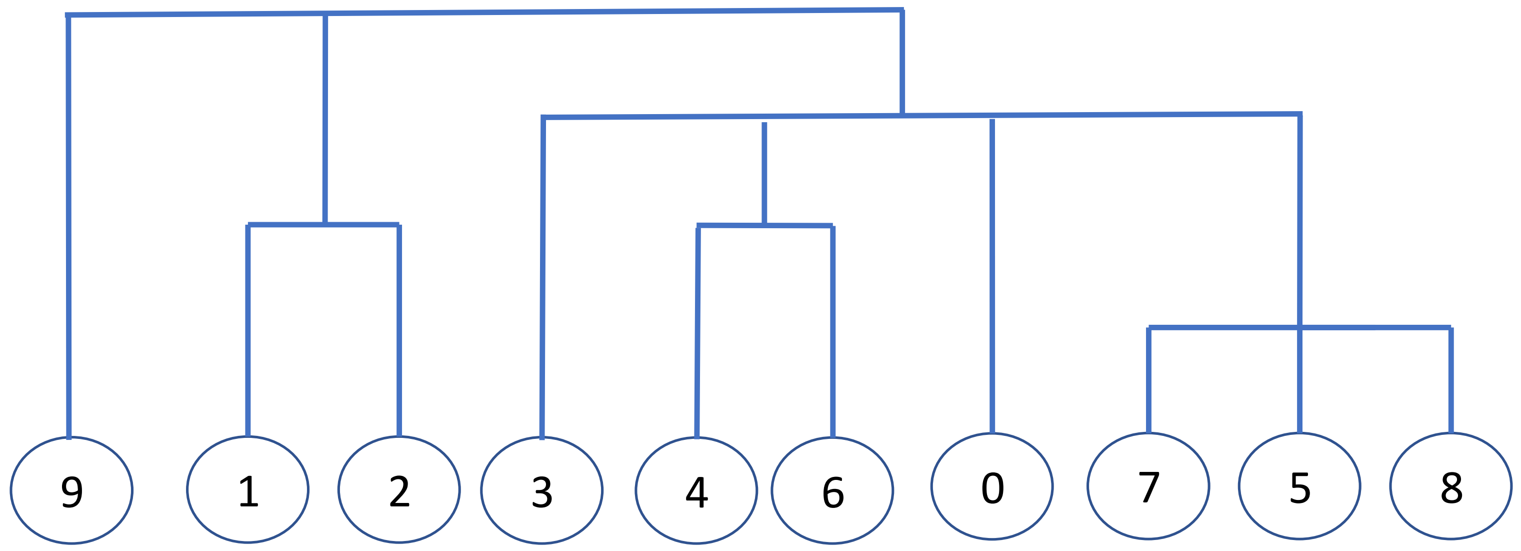
Example



	1,2,3,4,6 ,0,7,5,8	9
1,2,3,4,6 ,0,7,5,8		
9	4	

Example

- The final dendrogram is:



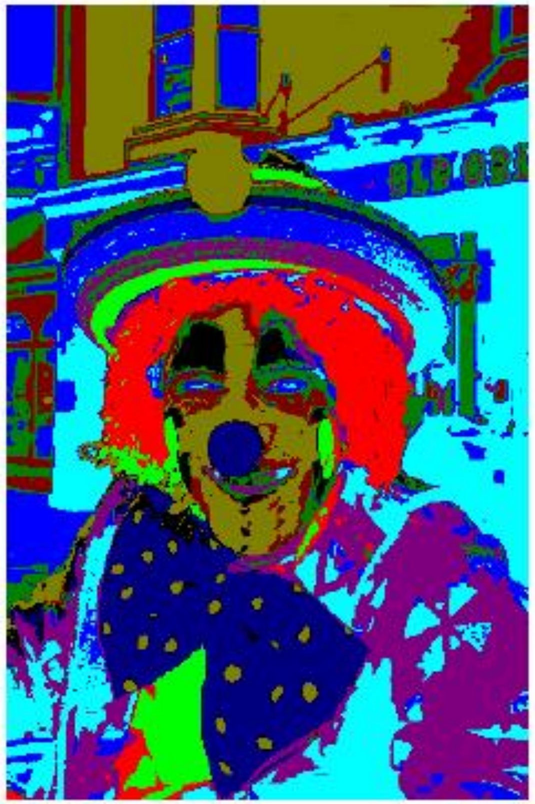
Unsupervised Learning – Summary

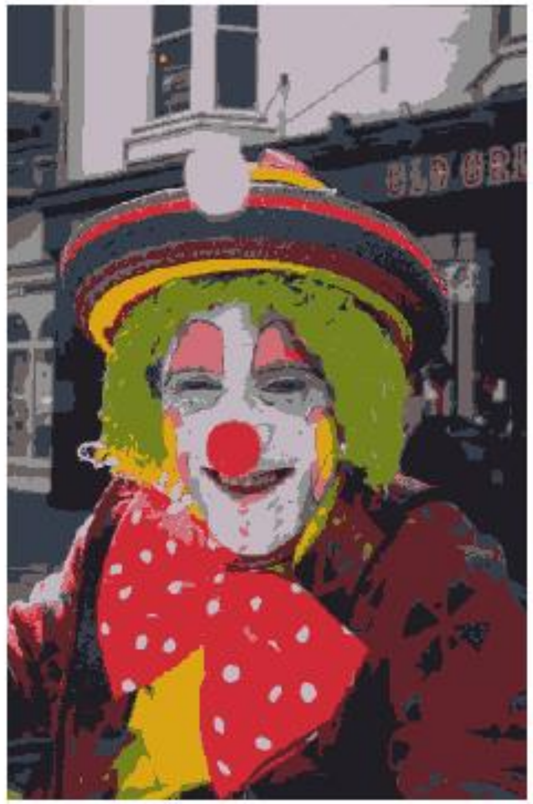
“The validation of clustering structures is the most difficult and frustrating part of cluster analysis.

Without a strong effort in this direction, cluster analysis will remain a black art accessible only to those true believers who have experience and great courage.”

Algorithms for Clustering Data, Jain and Dubes, 1988

Example – Image Compression





Questions

?