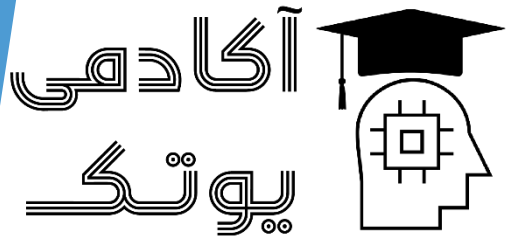


Machine Learning Course

Vahid Reza Khazaie

Unsupervised Learning

- **Dimensionality Reduction**
- **Clustering**
- **Autoencoding**
- **Anomaly Detection**
- **…**

Unsupervised Learning

- The vast majority of the available data is unlabeled: we have the input features **X**, but we do not have the labels **y**

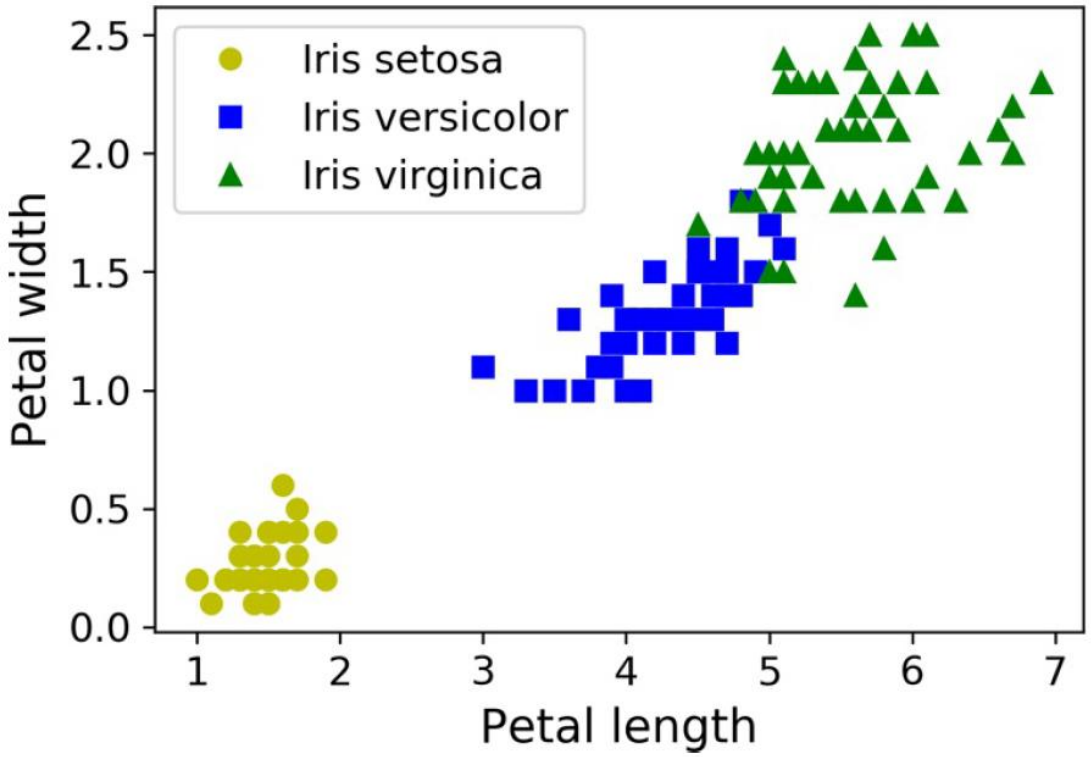
Unsupervised Learning

- **Clustering**

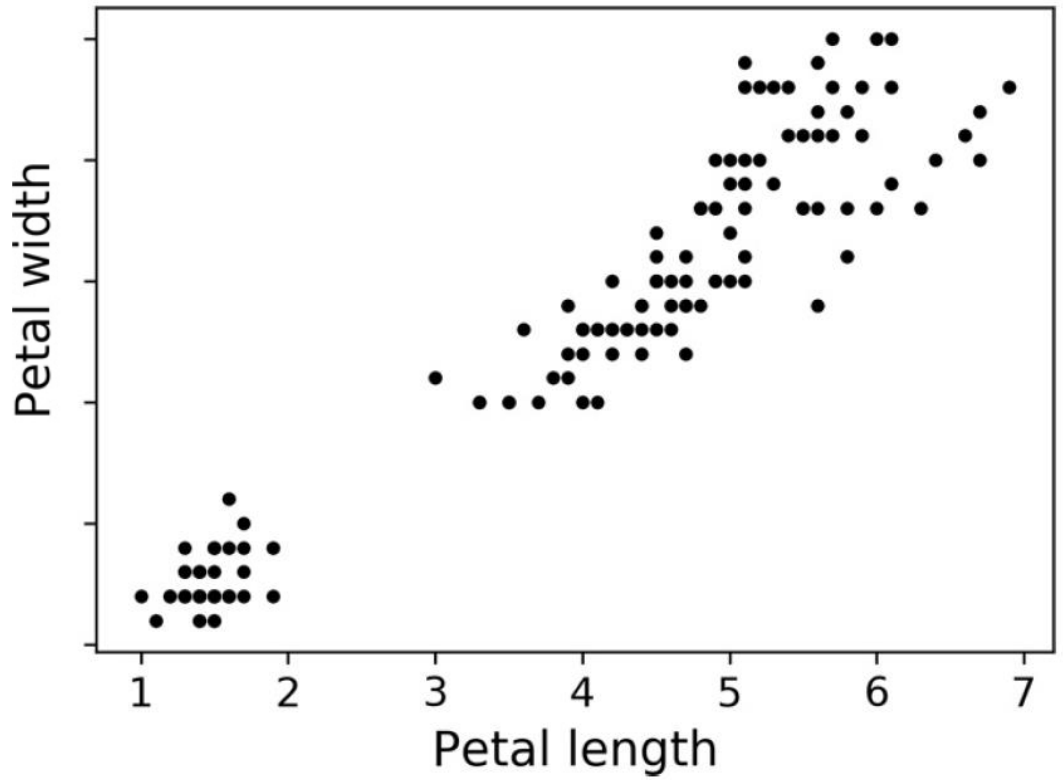
  It is the task of identifying similar instances and assigning them to *clusters*, or **groups of similar instances**.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task.

Unsupervised Learning



Unsupervised Learning



Clustering

- For customer segmentation

  You can cluster your customers based on their purchases and their activity on your website. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, customer segmentation can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.

Clustering

- For data analysis

  When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

Clustering

- As a dimensionality reduction technique

  Once a dataset has been clustered, it is usually possible to measure each instance's *affinity* with each cluster (affinity is any measure of how well an instance fits into a cluster). Each instance's feature vector **x** can then be replaced with the vector of its cluster affinities. If there are *k* clusters, then this vector is *k*-dimensional. This vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

Clustering

- For *anomaly detection* (also called *outlier detection*)

  Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second. Anomaly detection is particularly useful in detecting defects in manufacturing, or for *fraud detection*.

Clustering

- For semi-supervised learning

  If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

Clustering

- For search engines

  Some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database; similar images would end up in the same cluster. Then when a user provides a reference image, all you need to do is use the trained clustering model to find this image's cluster, and you can then simply return all the images from this cluster.

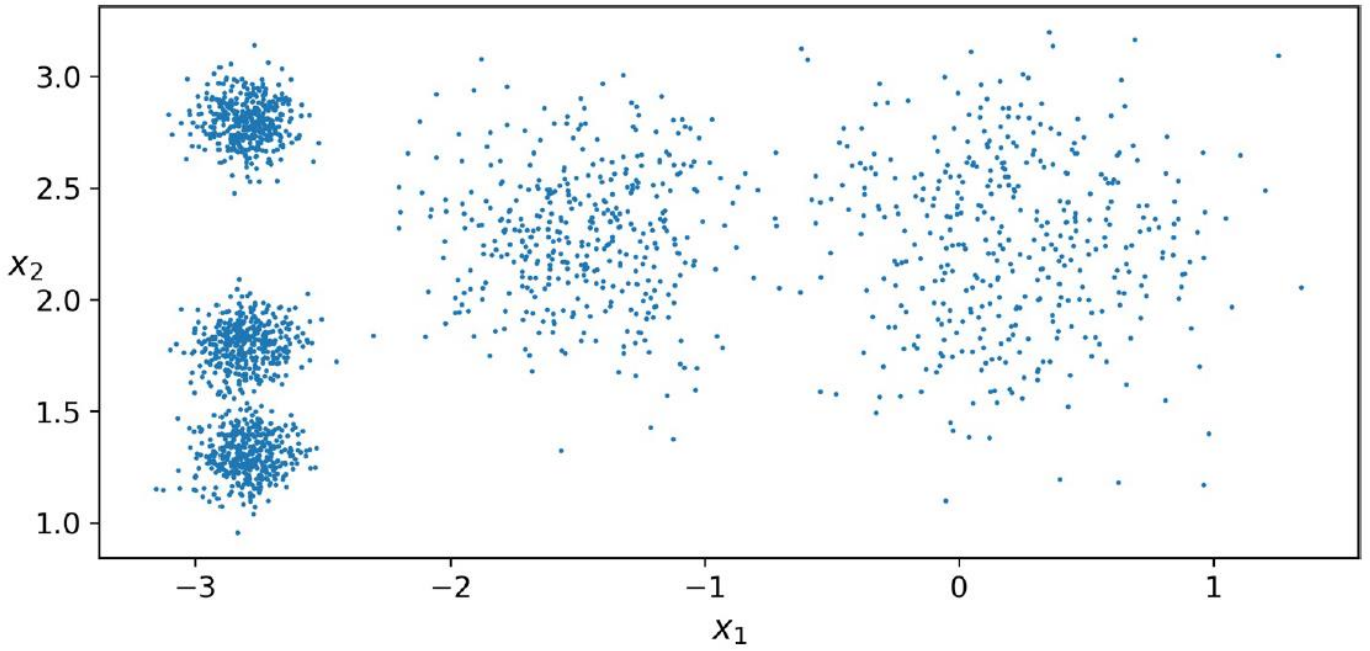
Clustering

- To segment an image

  By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in the image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

Clustering

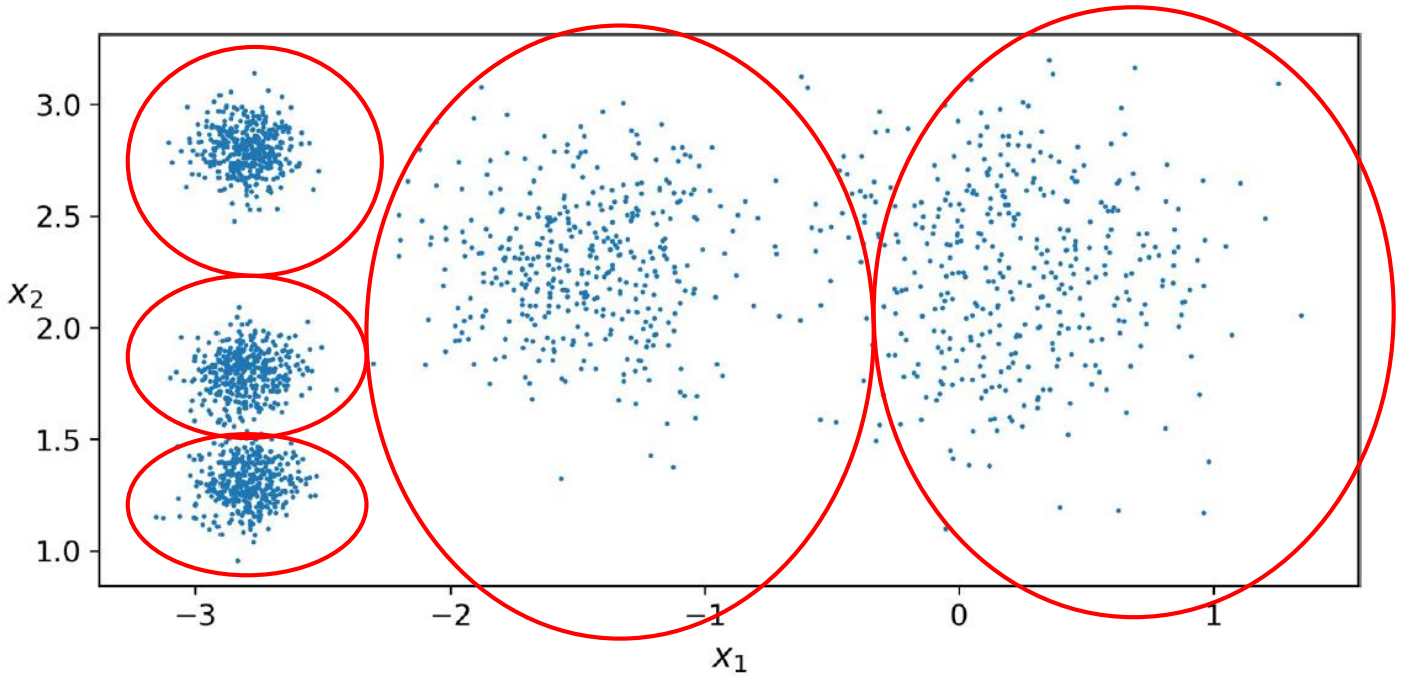
- K-Means



Clustering

- K-Means

  you can clearly see five blobs of instances. The K-Means algorithm is a simple algorithm capable of clustering this kind of dataset very quickly and efficiently, often in just a few iterations.



Clustering

- K-Means

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

Note that you have to specify the number of clusters *k* that the algorithm must find.

Clustering

- K-Means

  Each instance was assigned to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster that this instance gets assigned to by the algorithm: this is not to be confused with the class labels in classification (remember that clustering is an unsupervised learning task).

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

Clustering

- K-Means

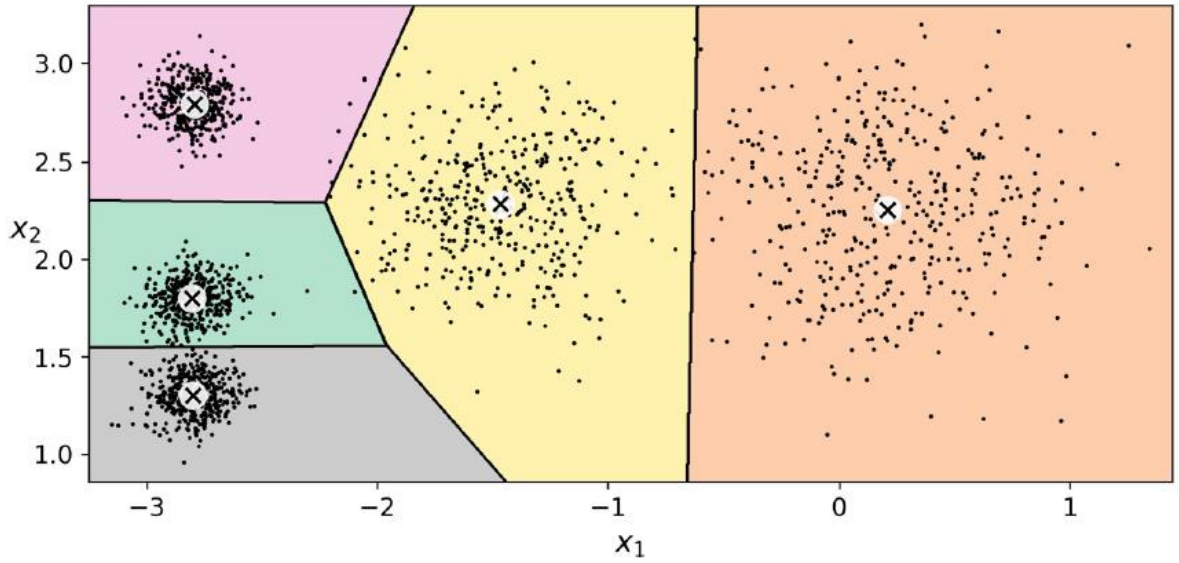
```
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

Clustering

- K-Means

  Instead of assigning each instance to a single cluster, which is called ***hard clustering***, it can be useful to give each instance a score per cluster, which is called ***soft clustering***. The score can be the distance between the instance and the centroid; conversely, it can be a similarity score.



Clustering

- K-Means

  In the Kmeans class, the transform() method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new)
array([[2.81093633, 0.32995317, 2.9042344 , 1.49439034, 2.88633901],
       [5.80730058, 2.80290755, 5.84739223, 4.4759332 , 5.84236351],
       [1.21475352, 3.29399768, 0.29040966, 1.69136631, 1.71086031],
       [0.72581411, 3.21806371, 0.36159148, 1.54808703, 1.21567622]])
```

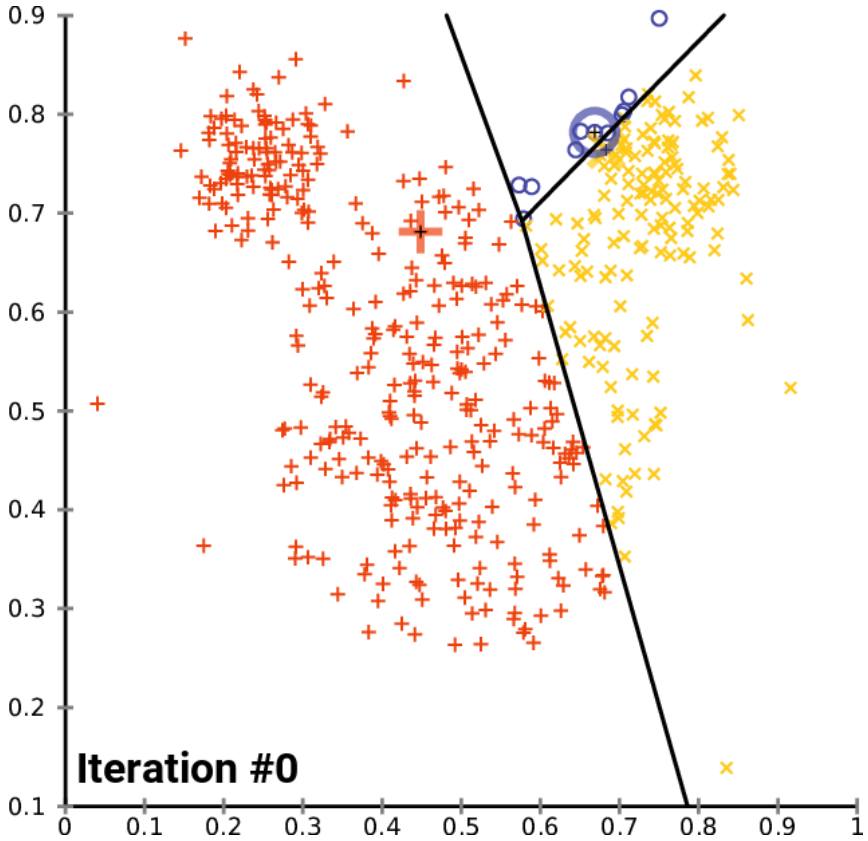
Clustering

- K-Means
  - Start by placing the centroids randomly (e.g., by picking *k* instances at random and using their locations as centroids).
  - Label the instances
  - Update the centroids
  - Do it until the centroids stop moving.

The algorithm is guaranteed to converge in a finite number of steps.

Clustering

- K-Means



Clustering

- K-Means

  Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization.

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, n_init=1)
```

Clustering

- K-Means

  Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the n_init hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier runs 10 times when you call fit(), and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model's *inertia*, which is the mean squared distance between each instance and its closest centroid.

```
>>> kmeans.inertia_
211.59853725816856
```

```
>>> kmeans.score(X)
-211.59853725816856
```

Clustering

- K-Means
  - K-Means++

    A smarter initialization step that tends to select centroids that are distant from one another, and this improvement makes the K-Means algorithm much less likely to converge to a suboptimal solution.

  - Accelerated K-Means

    Accelerates the algorithm by avoiding many unnecessary distance calculations. Achieved this by keeping track of lower and upper bounds for distances between instances and centroids.

  - mini-batch K-Means

    Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm typically by a factor of three or four and makes it possible to cluster huge datasets that do not fit in memory.

Clustering

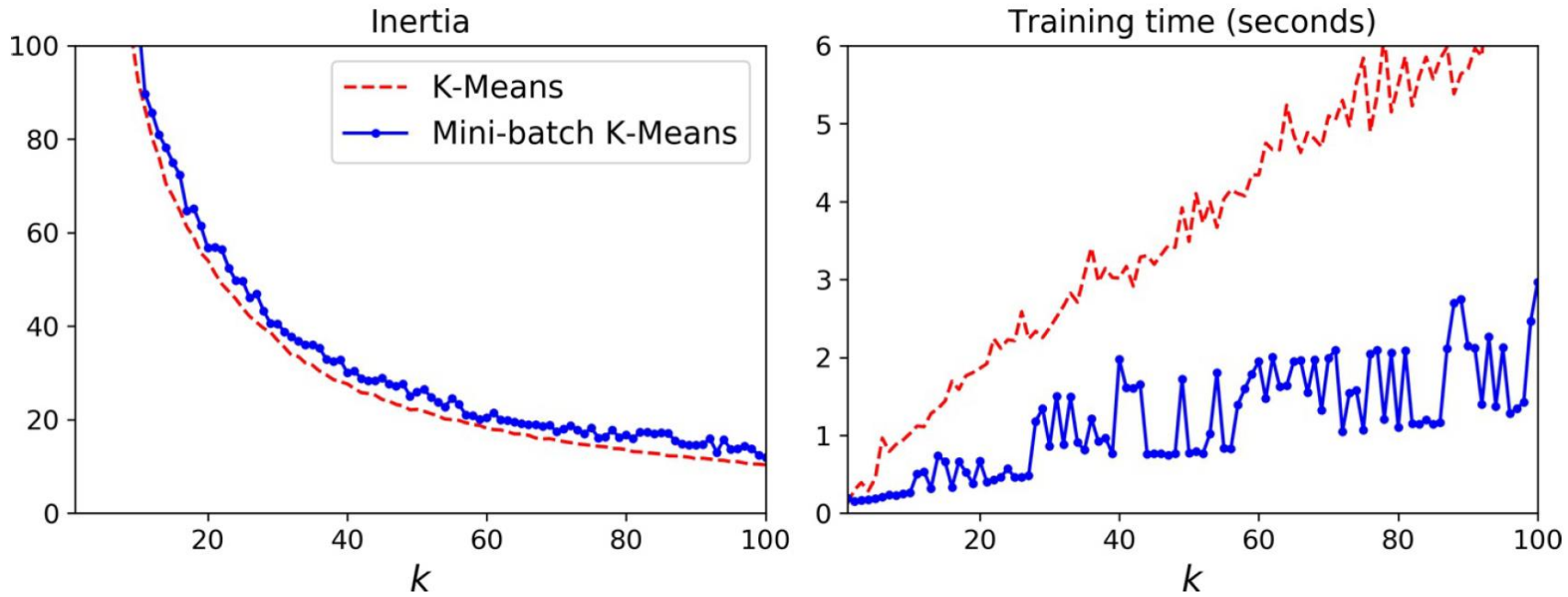
- K-Means
  - mini-batch K-Means

```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

Clustering

- K-Means
  - mini-batch K-Means



Clustering

- K-Means
  - **Optimal number of clusters**

    In general, it will not be so easy to know how to set *k*, and the result might be quite bad if you set it to the wrong value.

    The model with the lowest inertia?

Clustering

- K-Means
  - **Optimal number of clusters**

    In general, it will not be so easy to know how to set *k*, and the result might be quite bad if you set it to the wrong value.

    The model with the lowest inertia?

The inertia is not a good performance metric when trying to choose *k* because 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

Clustering

- K-Means
  - **Optimal number of clusters**

    In general, it will not be so easy to know how to set *k*, and the result might be quite bad if you set it to the wrong value.

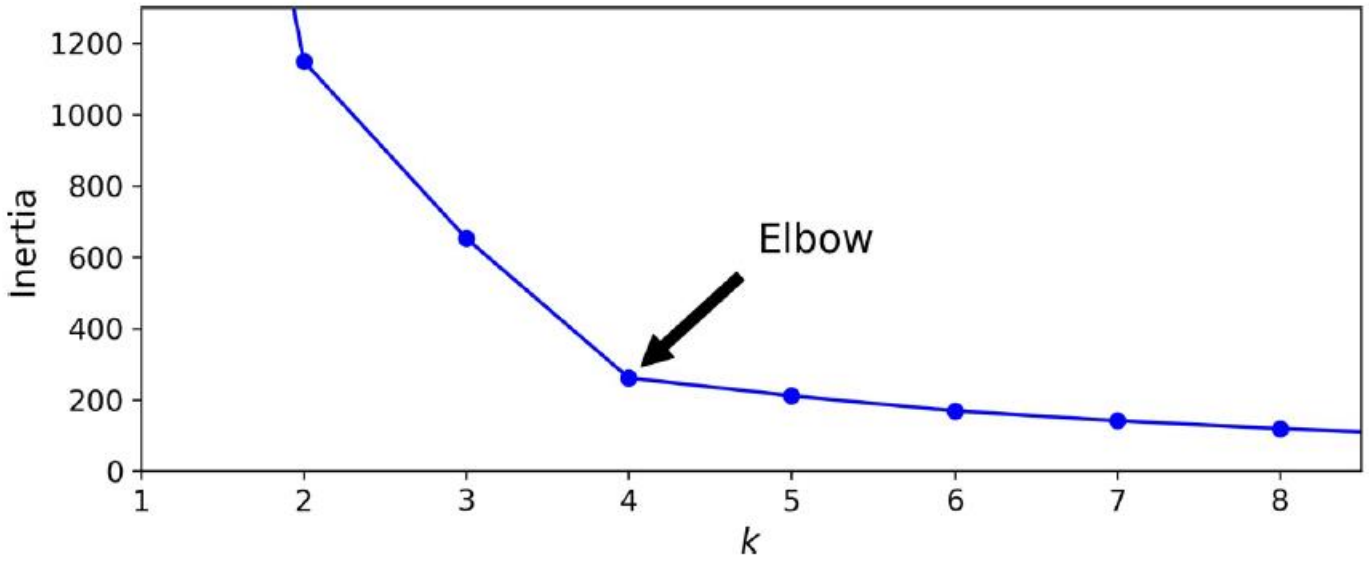
    The model with the lowest inertia?

The inertia is not a good performance metric when trying to choose *k* because 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

Clustering

- K-Means

As you can see, the inertia drops very quickly as we increase *k* up to 4, but then it decreases much more slowly as we keep increasing *k*. Any lower value than 4 would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.



Clustering

- K-Means

  Silhouette score

  which is the mean *silhouette coefficient* over all the instances.

  instance's silhouette coefficient = $(b - a) / \max(a, b)$

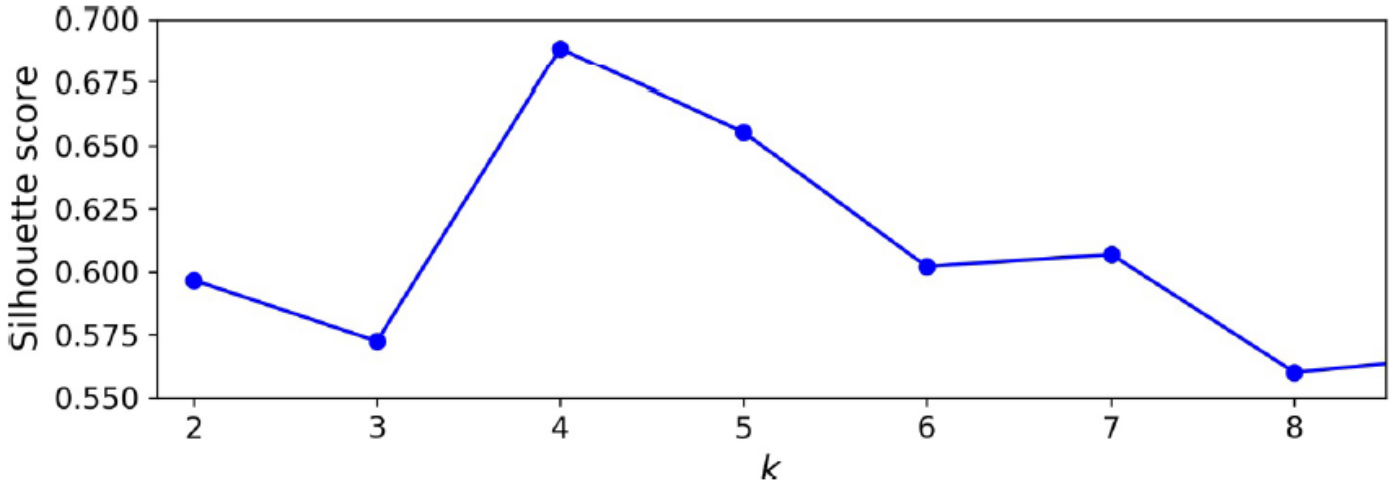
  a is the mean distance to the other instances in the same cluster and *b* is the mean nearest-cluster distance

The silhouette coefficient can vary between –1 and +1. A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to –1 means that the instance may have been assigned to the wrong cluster.

Clustering

- K-Means

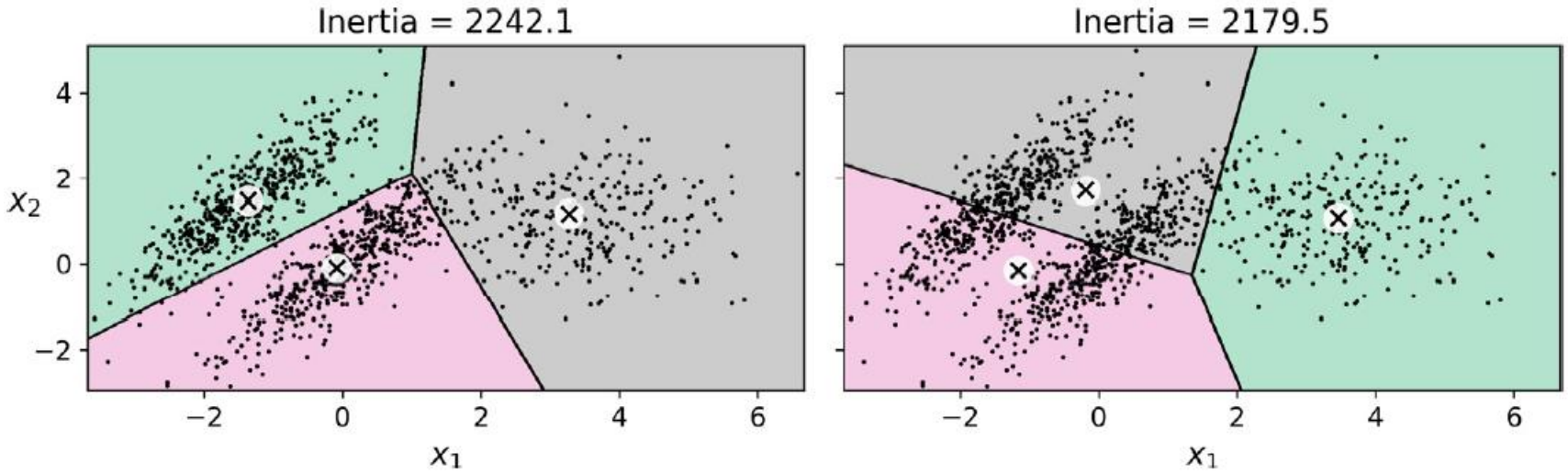
```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```



Clustering

- Limitations of K-Means

Despite its many merits, most notably being fast and scalable, K-Means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters. Moreover, K-Means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes.



It is important to scale the input features before you run K-Means, or the clusters may be very stretched and Kmeans will perform poorly

Clustering

- **DBSCAN**

  This algorithm defines clusters as continuous regions of high density. Here is how it works:

  - For each instance, the algorithm counts how many instances are located within a small distance ε (epsilon) from it. This region is called the instance's *ε-neighborhood*.
  - If an instance has at least min_samples instances in its ε-neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
  - All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
  - Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

Clustering

- **DBSCAN**

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

```
>>> dbscan.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0, ...,  3,  2,  3,  3,  4,  2,  6,  3])

>>> len(dbscan.core_sample_indices_)
808
>>> dbscan.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, ..., 992, 993, 995, 997, 998, 999])
>>> dbscan.components_
array([[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
                  ...
       [-0.94355873,  0.3278936 ],
       [ 0.79419406,  0.60777171]])
```