

# Module 6: Unsupervised Learning Techniques

**Video: 670\_mod6\_vid1**

Most machine learning applications today are based on supervised learning, but the vast majority of the available data is unlabeled. That is, we have the input features  $X$ , but we do not have the labels  $y$ .

Say you want to create a system that will take a few pictures of each item on a manufacturing production line and detect which items are defective. You can fairly easily create a system that will take pictures automatically, and this might give you thousands of pictures every day. You can then build a reasonably large dataset in just a few weeks.

But the problem is that there are no labels! If you want to train a regular binary classifier that will predict whether an item is defective or not, you will need to label every single picture as “defective” or “normal.” This will likely require human experts to sit down and manually go through all the pictures. This is a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. As a result, the labeled dataset will be quite small, and the classifier’s performance will be disappointing. Moreover, every time the company makes any change to its products, the whole process will need to be started over from scratch.

Wouldn’t it be great if the algorithm could just exploit the unlabeled data without needing humans to label every picture? This is where unsupervised learning comes in.

In this module, we will discuss unsupervised learning algorithms used for **clustering**. The goal in a clustering algorithm is to group similar instances together into **clusters**. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Specifically, we will study the ***K-Means*** and ***DBSCAN*** clustering algorithms.

## Clustering

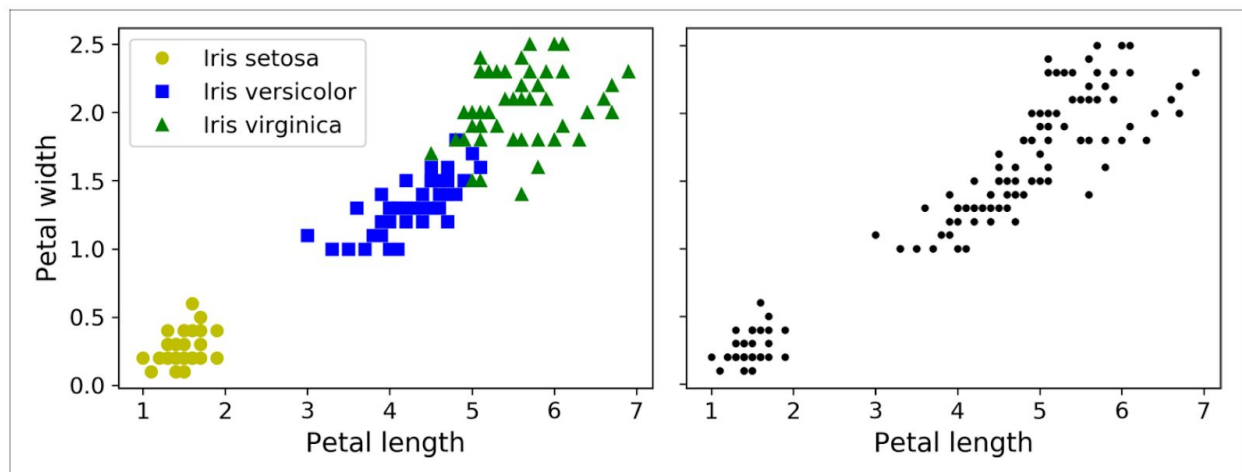
Suppose you are hiking in the mountains, and you stumble upon a plant you have never seen before. You look around and you notice a few more. They are not identical, yet they are sufficiently similar for you to know that they most likely belong to the same species (or at least the same genus). You may need a botanist to tell you what species that is, but you certainly don’t need an expert to identify groups of similar-looking objects.

This is called **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.

Let's consider the Iris dataset.

```
from sklearn.datasets import load_iris
data = load_iris()
X = data.data
y = data.target
data.target_names
```

Consider the following figure. On the left is the iris dataset (which was introduced in Module 4), where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as Logistic Regression, SVMs, or Random Forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore.



This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two distinct sub-clusters.

That said, the dataset has two additional features (sepal length and width), not represented here, and clustering algorithms can make good use of all features, so in fact there are algorithms that can identify the three clusters fairly well. For example, a Gaussian mixture model only assigns 5 instances out of 150 to the wrong cluster. We will not discuss Gaussian mixture models in depth in this course, but I will show you some results here:

```

from sklearn.mixture import GaussianMixture
y_pred = GaussianMixture(n_components=3, random_state=42).fit(X).predict(X)
mapping = np.array([2, 0, 1])
y_pred = np.array([mapping[cluster_id] for cluster_id in y_pred])

plt.plot(X[y_pred==0, 2], X[y_pred==0, 3], "yo", label="Cluster 1")
plt.plot(X[y_pred==1, 2], X[y_pred==1, 3], "bs", label="Cluster 2")
plt.plot(X[y_pred==2, 2], X[y_pred==2, 3], "g^", label="Cluster 3")
plt.xlabel("Petal length", fontsize=14)
plt.ylabel("Petal width", fontsize=14)
plt.legend(loc="upper left", fontsize=12)
plt.show()

np.sum(y_pred==y)
np.sum(y_pred==y) / len(y_pred)

```

Clustering is used in a wide variety of applications, including these:

#### *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.

#### *For data analysis*

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

#### *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.

#### *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.

### *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.

### *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.

### *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.

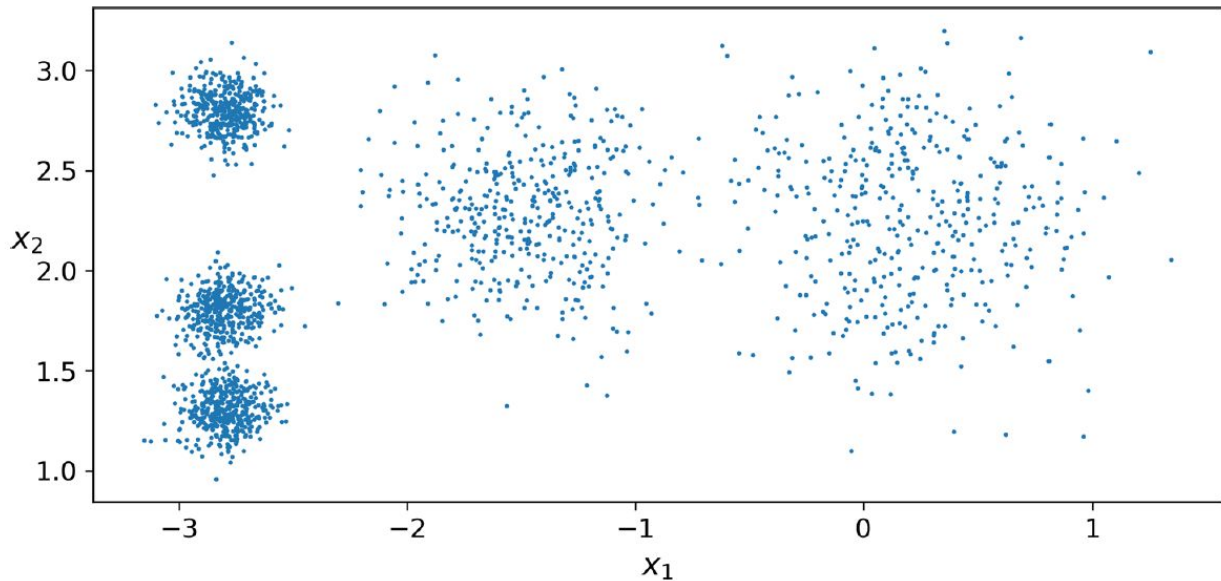
There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a *centroid*. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section, we will look at two popular clustering algorithms, ***K-Means*** and ***DBSCAN***, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

**Video: 670\_mod6\_vid2**

## K-Means

The K-Means algorithm is a simple algorithm capable of clustering data very quickly and efficiently, often in just a few iterations. It was proposed by Stuart Lloyd at Bell Labs in 1957 as a technique for pulse-code modulation, but it was only published outside of the company in 1982.



Let's start by generating some blob data:

```
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, random_state=7)
```

Now, let's plot that data:

```
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)
```

Clearly, this data is unlabeled, and you can clearly see five blobs of instances. Now, let's train a K-Means clusterer 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)
y_pred = kmeans.fit_predict(X)
```

Note that you have to specify the number of clusters  $k$  that the algorithm must find. In this example, it is pretty obvious from looking at the data that  $k$  should be set to 5, but in general it is not that easy. We will discuss this shortly.

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). The KMeans instance preserves a copy of the labels of the instances it was trained on, available via the `labels_` instance variable:

```
y_pred
y_pred is kmeans.labels_
```

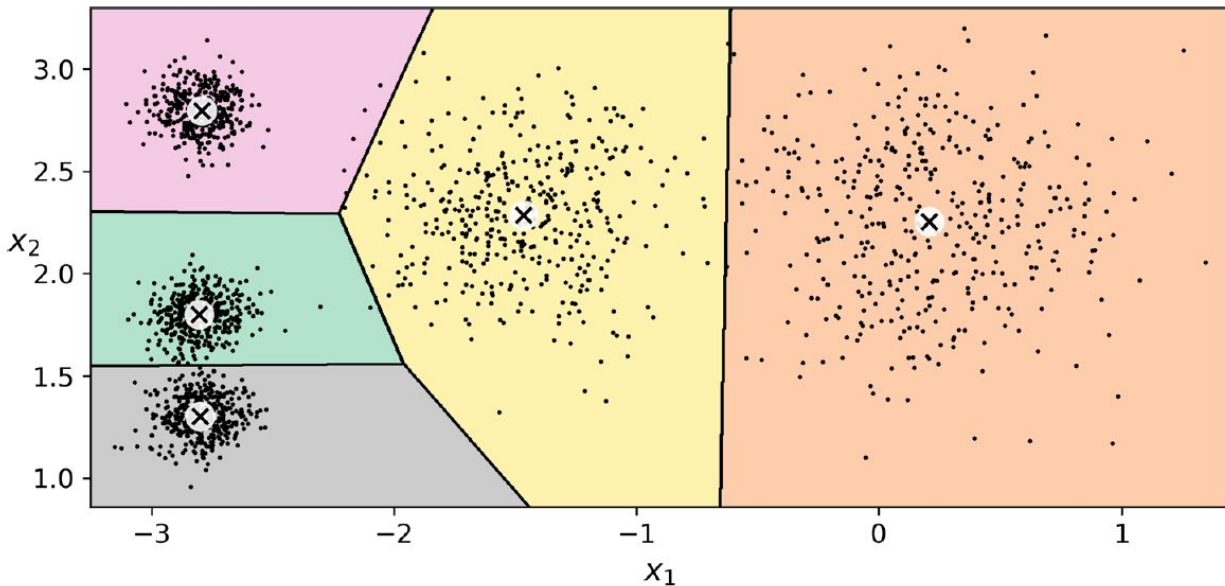
We can also take a look at the five centroids that the algorithm found:

```
kmeans.cluster_centers_
```

You can easily assign new instances to the cluster whose centroid is closest:

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

The following figure plots the cluster's decision boundaries.



The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled (especially near the boundary between the top-left cluster and the central cluster). Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

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 (or affinity), such as the Gaussian Radial Basis Function (introduced in Module 5). In the KMeans class, the `transform()` method measures the distance from each instance to every centroid:

```
kmeans.transform(X_new)
```

In this example, the first instance in `X_new` is located at a distance of 2.89 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid, 1.49 from the fourth centroid, and 2.81 from the fifth centroid. If you have a high-dimensional dataset and you transform it this way, you end up with a  $k$ -dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique.

## The K-Means Algorithm

So, how does the algorithm work? Well, suppose you were given the centroids. You could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate all the centroids by computing the mean of the instances for each cluster. But you are given neither the labels nor the centroids, so how can you proceed?

Well, just start by placing the centroids randomly (for example, by picking  $k$  instances at random and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small); it will not oscillate forever.

You can see the algorithm in action in the following figure.

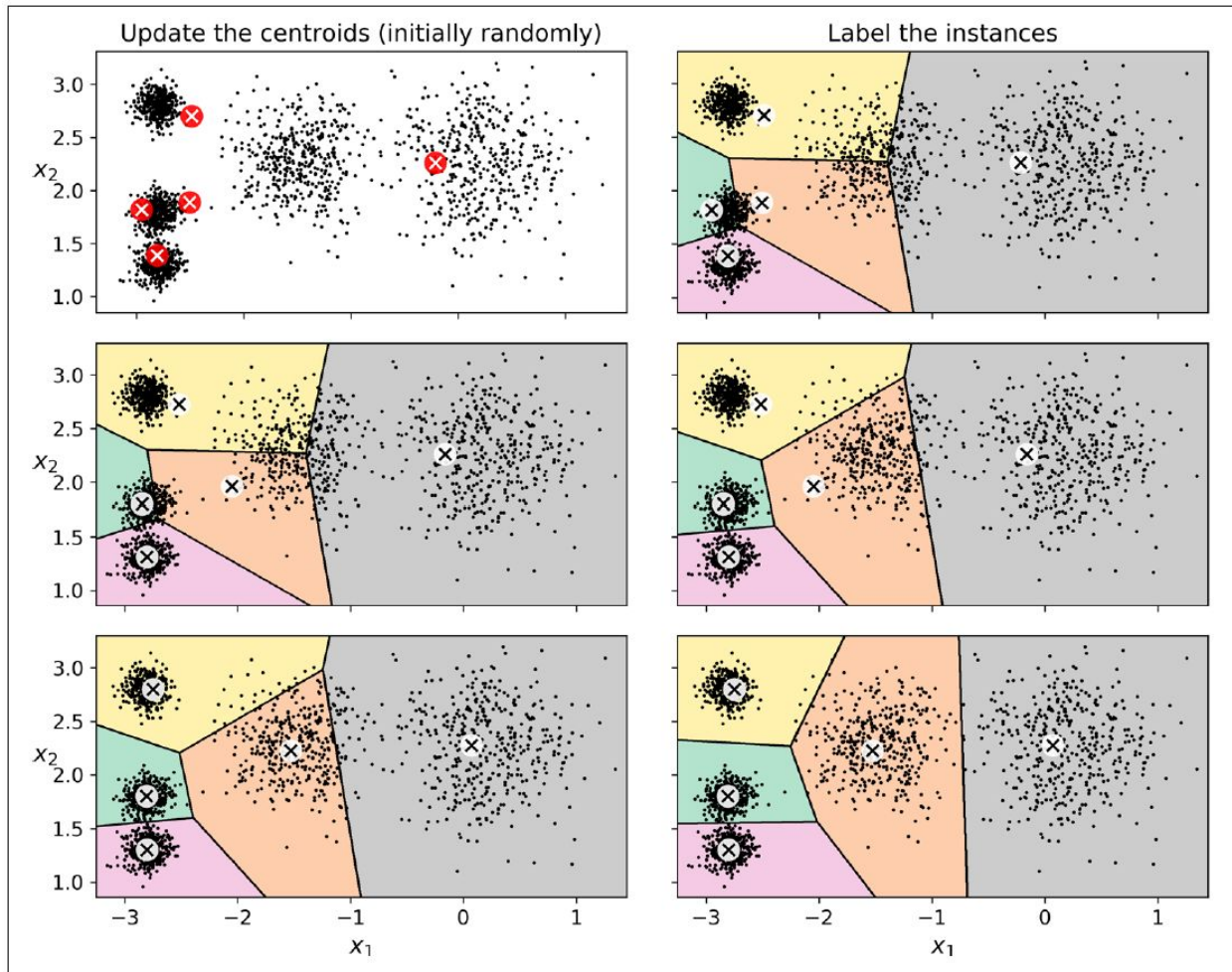
```
kmeans_iter1 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=1, random_state=1)
kmeans_iter2 = KMeans(n_clusters=5, init="random", n_init=1,
                      algorithm="full", max_iter=3, random_state=1)
kmeans_iter3 = KMeans(n_clusters=5, init="random",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

kmeans_iter1.fit(X)
kmeans_iter2.fit(X)
kmeans_iter3.fit(X)

plt.figure(figsize=(10, 8))
```

This figure illustrates six subplots, and each row of subplots corresponds to a K-Means algorithm that was run for 1, 3, and 5 iterations, respectively. The centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can see, by 5 iterations, the algorithm has reached a clustering that seems pretty good.





The computational complexity of the algorithm is generally linear with regard to the number of instances  $m$ , the number of clusters  $k$ , and the number of dimensions  $n$ . However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase exponentially with the number of instances. In practice, this rarely happens, and K-Means is generally one of the fastest clustering algorithms.

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. The following figure shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

```
kmeans_rnd_init1 = KMeans(n_clusters=5, init="random", n_init=1,
                           algorithm="full", random_state=0)
kmeans_rnd_init2 = KMeans(n_clusters=5, init="random", n_init=1,
                           algorithm="full", random_state=42)

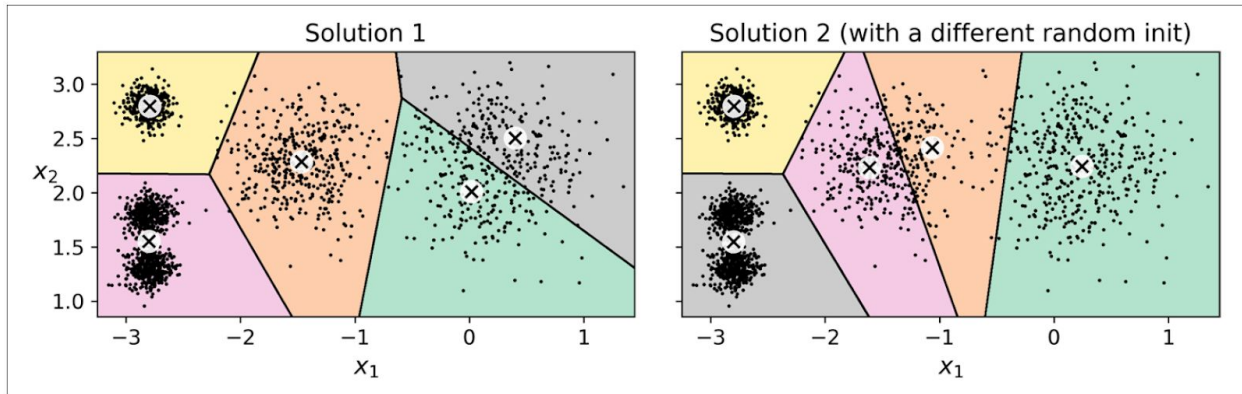
plot_clusterer_comparison(kmeans_rnd_init1, kmeans_rnd_init2, X,
```

```

        "Solution 1", "Solution 2 (with a different random init)")

    save_fig("kmeans_variability_plot")
    plt.show()

```



Next, we will look at a few ways you can mitigate this risk by improving the centroid initialization.

**Video: 670\_mod6\_vid4**

## Centroid Initialization Methods

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of centroids, and set `n_init` to 1:

```

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, random_state=42)
kmeans.fit(X)
kmeans.inertia_

```

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 ScikitLearn 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_rnd_init1.inertia_
kmeans_rnd_init2.inertia_

```

The KMeans class runs the algorithm `n_init` times and keeps the model with the lowest inertia. To specify this explicitly, we could write:

```
kmeans_rnd_10_inits = KMeans(n_clusters=5, init="random",
                             n_init=10, algorithm="full", random_state=11)
kmeans_rnd_10_inits.fit(X)
```

As we have seen, a model's inertia is accessible via the `inertia_` instance variable:

```
kmeans.inertia_
```

The `score()` method returns the negative inertia. Why negative? Because a predictor's `score()` method must always respect Scikit-Learn's "greater is better" rule: if a predictor is better than another, its `score()` method should return a greater score.

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

## K-Means++

An important improvement to the K-Means algorithm, called K-Means++, was proposed in a 2006 paper. The authors introduced 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. They showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution. Here is the K-Means++ initialization algorithm:

1. Take one centroid  $c_i$  chosen uniformly at random from the dataset.
2. Take a new centroid  $c_i$ , choosing an instance  $x$  with probability  $D(x_i)^2 / \sum_j 1^m D(x_j)^2$ , where  $D(x_i)$  is the distance between the instance  $x_i$  and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
3. Repeat the previous step until all  $k$  centroids have been chosen.

The KMeans class uses this initialization method by default. If you want to force it to use the original method (i.e., picking  $k$  instances randomly to define the initial centroids), then you can set the `init` hyperparameter to "random". You will rarely need to do this.

## Accelerated K-Means and Mini-Batch K-Means

Another important improvement to the K-Means algorithm was proposed in a 2003 paper. It considerably accelerates the algorithm by avoiding many unnecessary distance calculations. This is achieved by exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points) and by keeping track of lower and upper bounds for distances between instances and centroids. This is the algorithm the KMeans class uses by default (you can force it to use the original algorithm by setting the algorithm hyperparameter to "full", although you probably will never need to).

```
%timeit -n 50 KMeans(algorithm="elkan").fit(X)
%timeit -n 50 KMeans(algorithm="full").fit(X)
```

Yet another important variant of the K-Means algorithm was proposed in a 2010 paper. 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. Scikit-Learn implements this algorithm in the MiniBatchKMeans class. You can just use this class like the KMeans class:

```
from sklearn.cluster import MiniBatchKMeans

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

If the dataset does not fit in memory, the simplest option is to use the memmap class. Alternatively, you can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself.

```
import urllib
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1)
mnist.target = mnist.target.astype(np.int64)

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    mnist["data"], mnist["target"], random_state=42)
```

```

filename = "my_mnist.data"
X_mm = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X_mm[:] = X_train

minibatch_kmeans = MiniBatchKMeans(n_clusters=10, batch_size=10,
random_state=42)
minibatch_kmeans.fit(X_mm)

def load_next_batch(batch_size):
    return X[np.random.choice(len(X), batch_size, replace=False)]
k = 5
n_init = 10
n_iterations = 100
batch_size = 100
init_size = 500 # more data for K-Means++ initialization
evaluate_on_last_n_iters = 10

best_kmeans = None

for init in range(n_init):
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, init_size=init_size)
    X_init = load_next_batch(init_size)
    minibatch_kmeans.partial_fit(X_init)

    minibatch_kmeans.sum_inertia_ = 0
    for iteration in range(n_iterations):
        X_batch = load_next_batch(batch_size)
        minibatch_kmeans.partial_fit(X_batch)
        if iteration >= n_iterations - evaluate_on_last_n_iters:
            minibatch_kmeans.sum_inertia_ += minibatch_kmeans.inertia_

    if (best_kmeans is None or
        minibatch_kmeans.sum_inertia_ < best_kmeans.sum_inertia_):
        best_kmeans = minibatch_kmeans

best_kmeans.score(X)

```

Although the Mini-batch K-Means algorithm is much faster than the regular K-Means algorithm, its inertia is generally slightly worse, especially as the number of clusters increases. You can see this in the following figure.

```

from timeit import timeit

WARNING: The following cell will take 10 - 20 minutes to run!

times = np.empty((100, 2))
inertias = np.empty((100, 2))
for k in range(1, 101):
    kmeans_ = KMeans(n_clusters=k, random_state=42)
    minibatch_kmeans = MiniBatchKMeans(n_clusters=k, random_state=42)
    print("\r{}/{}".format(k, 100), end="")
    times[k-1, 0] = timeit("kmeans_.fit(X)", number=10, globals=globals())
    times[k-1, 1] = timeit("minibatch_kmeans.fit(X)", number=10, globals=globals())
    inertias[k-1, 0] = kmeans_.inertia_

```

```

inertias[k-1, 1] = minibatch_kmeans.inertia_

plt.figure(figsize=(10,4))

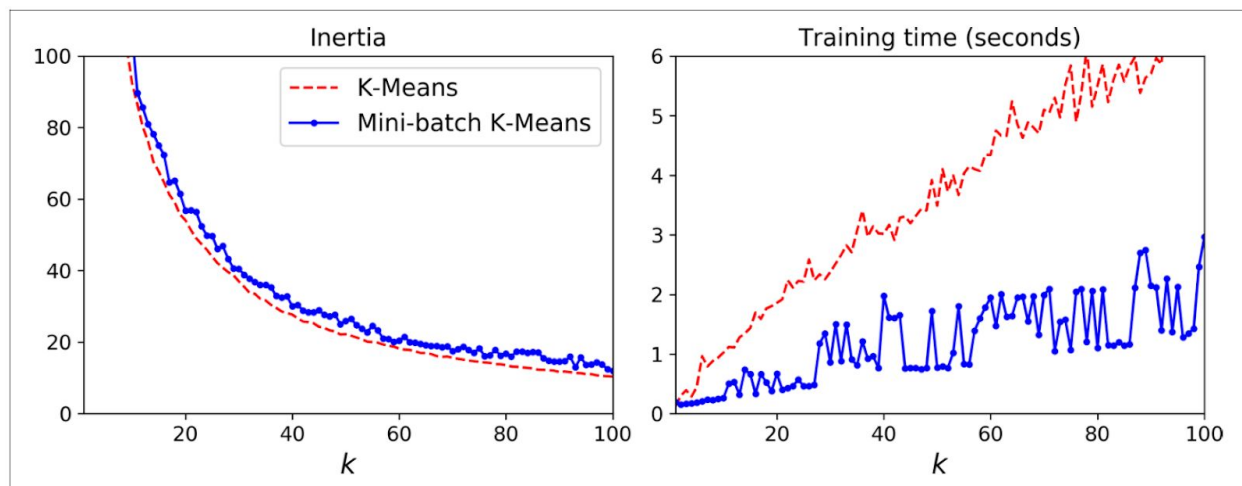
plt.subplot(121)
plt.plot(range(1, 101), inertias[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), inertias[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Inertia", fontsize=14)
plt.legend(fontsize=14)
plt.axis([1, 100, 0, 100])

plt.subplot(122)
plt.plot(range(1, 101), times[:, 0], "r--", label="K-Means")
plt.plot(range(1, 101), times[:, 1], "b.-", label="Mini-batch K-Means")
plt.xlabel("$k$", fontsize=16)
plt.title("Training time (seconds)", fontsize=14)
plt.axis([1, 100, 0, 6])

save_fig("minibatch_kmeans_vs_kmeans")
plt.show()

```

The plot on the left compares the inertias of Mini-batch K-Means and regular K-Means models trained on the previous dataset using various numbers of clusters  $k$ . The difference between the two curves remains fairly constant, but this difference becomes more and more significant as  $k$  increases, since the inertia becomes smaller and smaller. In the plot on the right, you can see that Mini-batch K-Means is much faster than regular K-Means, and this difference increases with  $k$ .



**Video: 670\_mod6\_vid6**

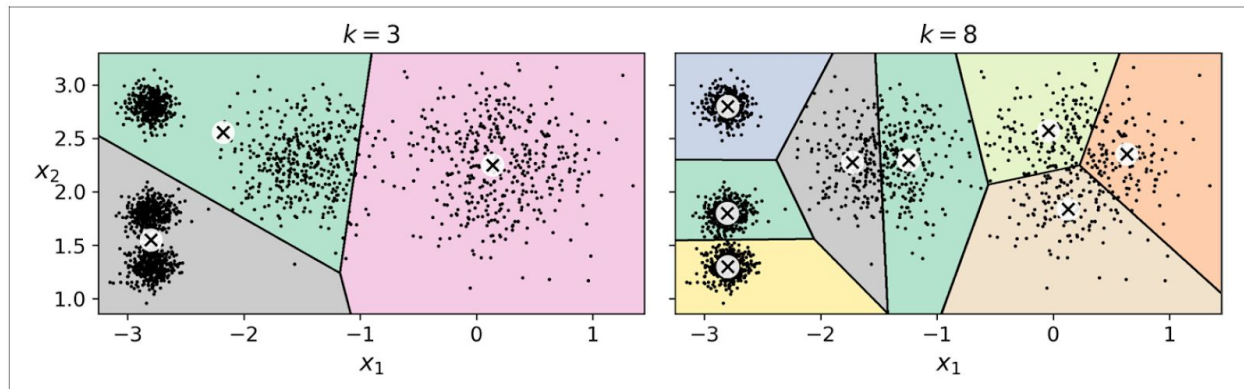
## Finding the Optimal Number of Clusters

So far, we have set the number of clusters  $k$  to 5 because it was obvious by looking at the data that this was the correct number of clusters. But 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. As you can see in the following figure, setting  $k$  to 3 or 8 results in fairly bad models.

```
kmeans_k3 = KMeans(n_clusters=3, random_state=42)
kmeans_k8 = KMeans(n_clusters=8, random_state=42)

plot_clusterer_comparison(kmeans_k3, kmeans_k8, X, "$k=3$", "$k=8$")
save_fig("bad_n_clusters_plot")
plt.show()
```



You might be thinking that we could just pick the model with the lowest inertia, right? Unfortunately, it is not that simple.

```
kmeans_k3.inertia_
kmeans_k8.inertia_
```

The inertia for  $k=3$  is 653.2, which is much higher than for  $k=5$  (which was 211.6). But with  $k=8$ , the inertia is just 118.4. 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. Let's plot the inertia as a function of  $k$ :

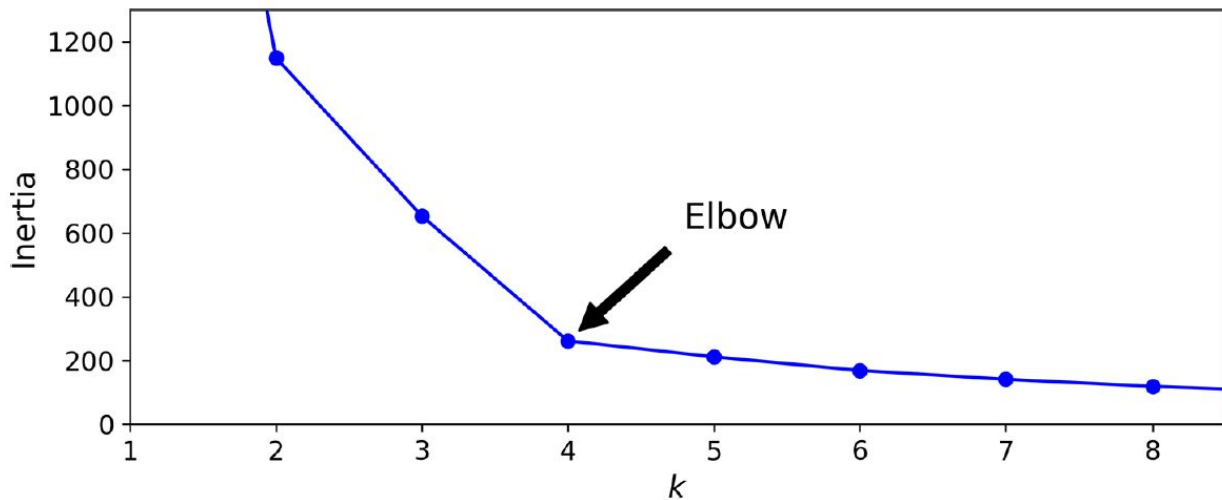
```
kmeans_per_k = [KMeans(n_clusters=k, random_state=42).fit(X)
                  for k in range(1, 10)]
inertias = [model.inertia_ for model in kmeans_per_k]

plt.figure(figsize=(8, 3.5))
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, 8.5, 0, 1300])
save_fig("inertia_vs_k_plot")
plt.show()

```



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  $k$ . This curve has roughly the shape of an arm, and there is an “elbow” at  $k = 4$ . So, if we didn't know better, 4 would be a good choice: any lower value 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.

```

plot_decision_boundaries(kmeans_per_k[4-1], X)
plt.show()

```

This technique for choosing the best value for the number of clusters is rather coarse. A more precise approach (but also more computationally expensive) is to use the ***silhouette score***, which is the mean ***silhouette coefficient*** over all the instances. An instance's silhouette coefficient is equal to  $(b - a) / \max(a, b)$ , where  $a$  is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and  $b$  is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes  $b$ , excluding the instance's own cluster). 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.

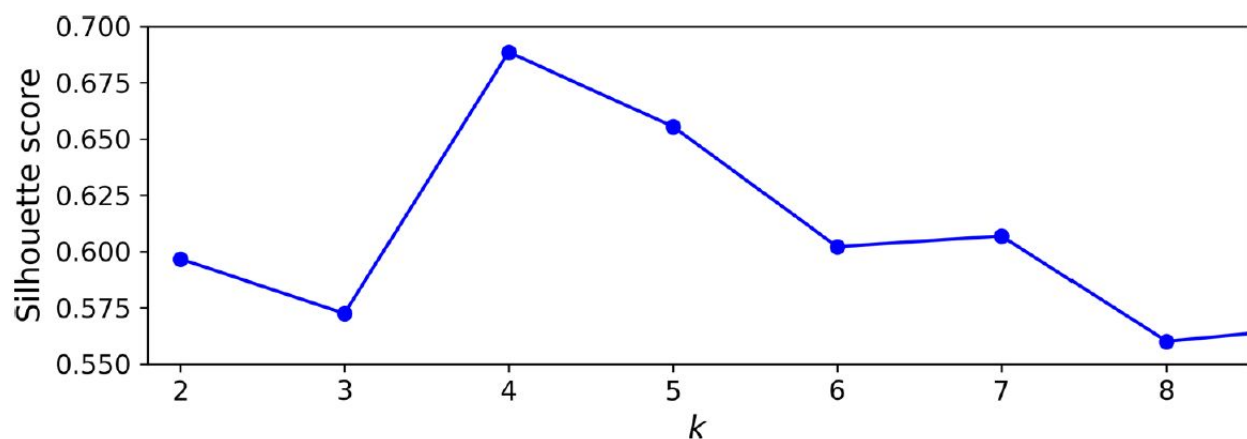


To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

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

silhouette_scores = [silhouette_score(X, model.labels_)
for model in kmeans_per_k[1:]]
```

Let's compare the silhouette scores for different numbers of clusters:



As you can see, this visualization is much richer than the previous one: although it confirms that  $k = 4$  is a very good choice, it also underlines the fact that  $k = 5$  is quite good as well, and much better than  $k = 6$  or  $7$ . This was not visible when we compared inertias.

An even more informative visualization is obtained when you plot every instance's silhouette coefficient, sorted by the cluster they are assigned to and by the value of the coefficient. This is called a ***silhouette diagram*** (see figure below). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances the cluster contains, and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better). The dashed line indicates the mean silhouette coefficient.

```
from sklearn.metrics import silhouette_samples
from matplotlib.ticker import FixedLocator, FixedFormatter

plt.figure(figsize=(11, 9))

for k in (3, 4, 5, 6):
    plt.subplot(2, 2, k - 2)

    y_pred = kmeans_per_k[k - 1].labels_
```

```

silhouette_coefficients = silhouette_samples(X, y_pred)

padding = len(X) // 30
pos = padding
ticks = []
for i in range(k):
    coeffs = silhouette_coefficients[y_pred == i]
    coeffs.sort()

    color = mpl.cm.Spectral(i / k)
    plt.fill_betweenx(np.arange(pos, pos + len(coeffs)), 0, coeffs,
                      facecolor=color, edgecolor=color, alpha=0.7)
    ticks.append(pos + len(coeffs) // 2)
    pos += len(coeffs) + padding

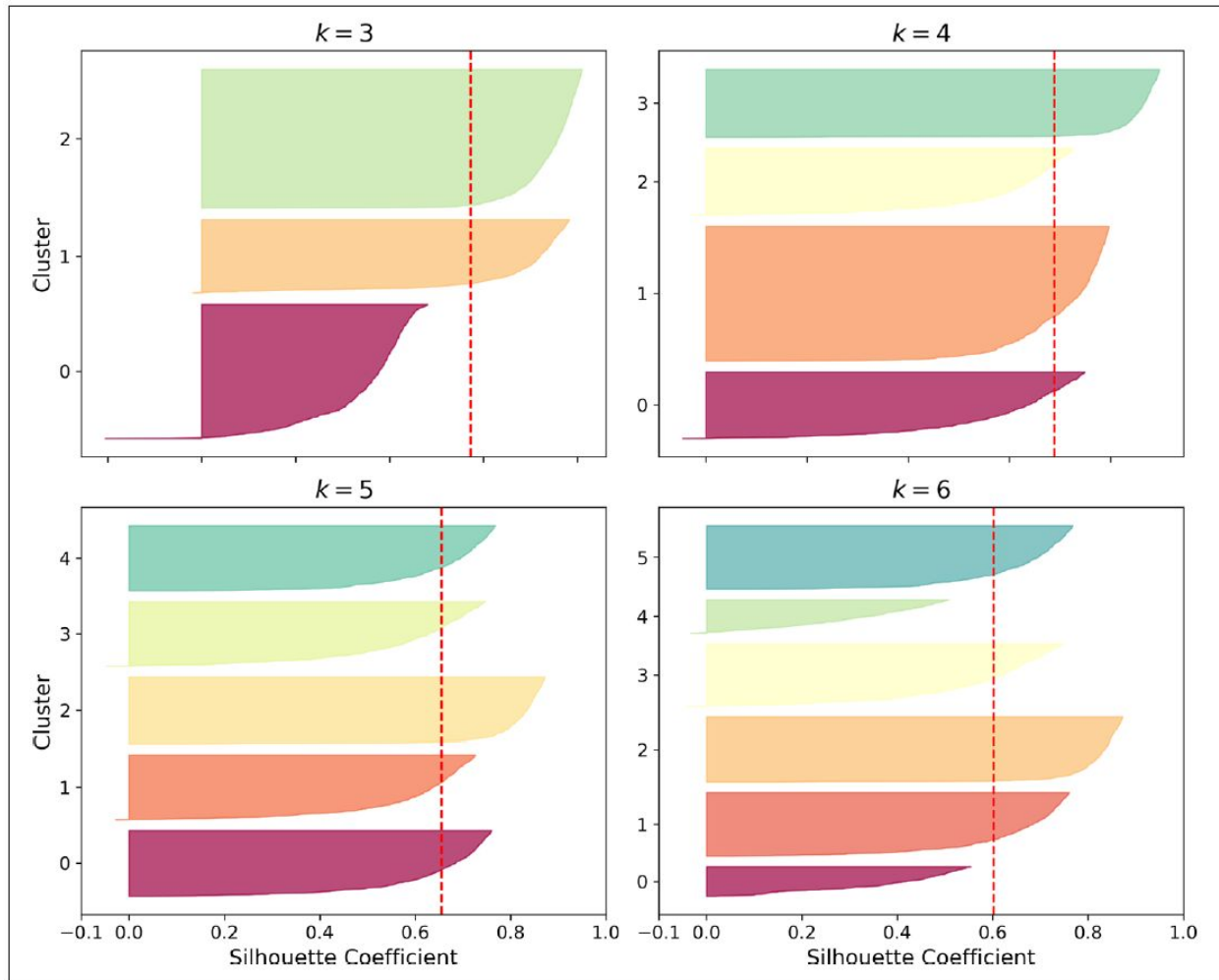
plt.gca().yaxis.set_major_locator(FixedLocator(ticks))
plt.gca().yaxis.set_major_formatter(FixedFormatter(range(k)))
if k in (3, 5):
    plt.ylabel("Cluster")

if k in (5, 6):
    plt.gca().set_xticks([-0.1, 0, 0.2, 0.4, 0.6, 0.8, 1])
    plt.xlabel("Silhouette Coefficient")
else:
    plt.tick_params(labelbottom=False)

plt.axvline(x=silhouette_scores[k - 2], color="red", linestyle="--")
plt.title("$k={}$".format(k), fontsize=16)

save_fig("silhouette_analysis_plot")
plt.show()

```



The vertical dashed lines represent the silhouette score for each number of clusters. When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. We can see that when  $k = 3$  and when  $k = 6$ , we get bad clusters. But when  $k = 4$  or  $k = 5$ , the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0. When  $k = 4$ , the cluster at index 1 (the third from the top) is rather big. When  $k = 5$ , all clusters have similar sizes. So, even though the overall silhouette score from  $k = 4$  is slightly greater than for  $k = 5$ , it seems like a good idea to use  $k = 5$  to get clusters of similar sizes.

**Video: 670\_mod6\_vid7**

## Limits 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, which can be quite a hassle. Moreover, K-Means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes. For example, the following figure shows how K-Means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities, and orientations.

Let's start by making some blobs:

```
X1, y1 = make_blobs(n_samples=1000, centers=((4, -4), (0, 0)), random_state=42)
X1 = X1.dot(np.array([[0.374, 0.95], [0.732, 0.598]]))
X2, y2 = make_blobs(n_samples=250, centers=1, random_state=42)
X2 = X2 + [6, -8]
X = np.r_[X1, X2]
y = np.r_[y1, y2]

plot_clusters(X)
```

Next, let's fit two different K-Means models:

```
kmeans_good = KMeans(n_clusters=3, init=np.array([[-1.5, 2.5], [0.5, 0], [4, 0]]),
                    n_init=1, random_state=42)
kmeans_bad = KMeans(n_clusters=3, random_state=42)
kmeans_good.fit(X)
kmeans_bad.fit(X)
```

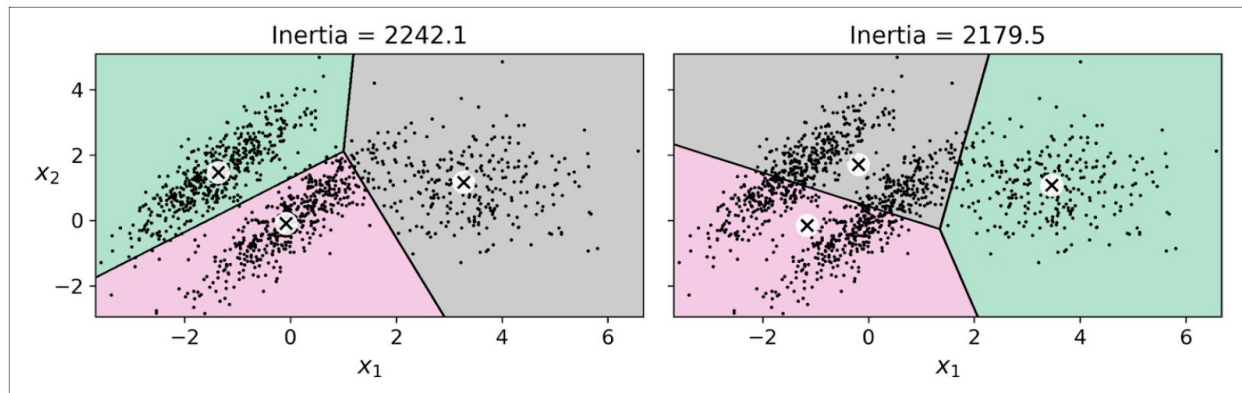
Finally, we can plot the decision boundaries for the data:

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

plt.subplot(121)
plot_decision_boundaries(kmeans_good, X)
plt.title("Inertia = {:.1f}".format(kmeans_good.inertia_), fontsize=14)

plt.subplot(122)
plot_decision_boundaries(kmeans_bad, X, show_ylabels=False)
plt.title("Inertia = {:.1f}".format(kmeans_bad.inertia_), fontsize=14)

save_fig("bad_kmeans_plot")
plt.show()
```



As you can see, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, Gaussian mixture models work great.

It is important to scale the input features before you run K-Means, or the clusters may be very stretched and K-Means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally improves things.

Now let's look at a few ways we can benefit from clustering. We will use K-Means, but feel free to experiment with other clustering algorithms.

**Video: 670\_mod6\_vid8**

## Using Clustering for Image Segmentation

Image segmentation is the task of partitioning an image into multiple segments. In **semantic segmentation**, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians). In **instance segmentation**, all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian. The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks (which we will study in ML II). Here, we are going to do something much simpler: color segmentation. We will simply assign pixels to the same segment if they have a similar color. In some applications, this may be sufficient. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.

Let's download an image and then use Matplotlib's `imread()` function to load the image (this is the upper-left image in the following figure):

```
# Download the ladybug image
images_path = os.path.join(PROJECT_ROOT_DIR, "images", "unsupervised_learning")
os.makedirs(images_path, exist_ok=True)
DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml2/master/"
filename = "ladybug.png"
print("Downloading", filename)
url = DOWNLOAD_ROOT + "images/unsupervised_learning/" + filename
urllib.request.urlretrieve(url, os.path.join(images_path, filename))

from matplotlib.image import imread # or `from imageio import imread`
image = imread(os.path.join
                ("images", "unsupervised_learning", "ladybug.png"))
image.shape
```

Let's take a look at the image:

```
plt.figure(figsize=(8,6))
plt.imshow(image)
save_fig('ladybugimage')
plt.show()
```

The image is represented as a 3D array. The first dimension's size is the height; the second is the width; and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue, each between 0.0 and 1.0 (or between 0 and 255, if you use `imageio.imread()`). Some images may have fewer channels, such as grayscale images (one channel). And some images may have more channels, such as images with an additional alpha channel for transparency or satellite images, which often contain channels for many light frequencies (e.g., infrared). The following code reshapes the array to get a long list of RGB colors, then it clusters these colors using K-Means:

```
X = image.reshape(-1, 3)
kmeans = KMeans(n_clusters=8).fit(X)
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

For example, it may identify a color cluster for all shades of green. Next, for each color (e.g., dark green), it looks for the mean color of the pixel's color cluster. For example, all shades of green may be replaced with the same light green color (assuming the mean color of the green

cluster is light green). Finally, it reshapes this long list of colors to get the same shape as the original image. And we're done!

This outputs the image shown in the upper right of the following figure:

```
segmented_imgs = []
n_colors = (10, 8, 6, 4, 2)
for n_clusters in n_colors:
    kmeans = KMeans(n_clusters=n_clusters, random_state=42).fit(X)
    segmented_img = kmeans.cluster_centers_[kmeans.labels_]
    segmented_imgs.append(segmented_img.reshape(image.shape))

plt.figure(figsize=(10,5))
plt.subplots_adjust(wspace=0.05, hspace=0.1)

plt.subplot(231)
plt.imshow(image)
plt.title("Original image")
plt.axis('off')

for idx, n_clusters in enumerate(n_colors):
    plt.subplot(232 + idx)
    plt.imshow(segmented_imgs[idx])

    plt.title("{} colors".format(n_clusters))
    plt.axis('off')

save_fig('image_segmentation_diagram', tight_layout=False)
plt.show()
```



You should experiment with various numbers of clusters on your own, as shown in the figure. When you use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because K-Means prefers clusters of similar sizes. The ladybug is small—much smaller than the rest of the image—so even though its color is flashy, K-Means fails to dedicate a cluster to it.

**Video: 670\_mod6\_vid9**

## Using Clustering for Preprocessing

Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm. As an example of using clustering for dimensionality reduction, let's tackle the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale  $8 \times 8$  images representing the digits 0 to 9. First, load the dataset:

```
from sklearn.datasets import load_digits
X_digits, y_digits = load_digits(return_X_y=True)
```

Now, split it into a training set and a test set:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
    train_test_split(X_digits, y_digits)
```

Next, fit a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train)
```

Let's evaluate its accuracy on the test set:

```
log_reg.score(X_test, y_test)
```

Alright, so that's our baseline: 96.9% accuracy. Let's see if we can do better by using K-Means as a preprocessing step. We will create a pipeline that will first cluster the training set into 50 clusters and replace the images with their distances to these 50 clusters, then apply a Logistic Regression model:

```
from sklearn.pipeline import Pipeline
```



```

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)

```

Since there are 10 different digits, it is tempting to set the number of clusters to 10. However, each digit can be written several different ways, so it is preferable to use a larger number of clusters, such as 50.

Now let's evaluate this classification pipeline:

```

pipeline.score(X_test, y_test)

```

How about that? We reduced the error rate by over 35%!

But we chose the number of clusters  $k$  arbitrarily; we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for  $k$  is much simpler than earlier. There's no need to perform silhouette analysis or minimize the inertia; the best value of  $k$  is simply the one that results in the best classification performance during cross-validation. We can use GridSearchCV to find the optimal number of clusters:

```

from sklearn.model_selection import GridSearchCV
param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)

```

Let's look at the best value for  $k$  and the performance of the resulting pipeline:

```

grid_clf.best_params_
grid_clf.score(X_test, y_test)

```

**Video: 670\_mod6\_vid10**

## Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. Let's train a Logistic Regression model on a sample of 50 labeled instances from the digits dataset:

```
n_labeled = 50
log_reg = LogisticRegression()
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

What is the performance of this model on the test set?

```
log_reg.score(X_test, y_test)
```

The accuracy is just 83.3%. It should come as no surprise that this is much lower than earlier, when we trained the model on the full training set. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then for each cluster, let's find the image closest to the centroid. We will call these images the representative images:

```
k = 50
kmeans = KMeans(n_clusters=k)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = np.argmin(X_digits_dist, axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

The following figure shows these 50 representative images:



Let's look at each image and manually label it:

```
y_representative_digits =
    np.array([4, 8, 0, 6, 8, 3, ..., 7, 6, 2, 3, 1, 1])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
log_reg = LogisticRegression()
log_reg.fit(X_representative_digits, y_representative_digits)
```

```
log_reg.score(X_test, y_test)
```

Wow! We jumped from 83.3% accuracy to 91.7%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int32)
for i in range(k):
    y_train_propagated[kmeans.labels_==i] =
        y_representative_digits[i]
```

Now let's train the model again and look at its performance:

```
log_reg = LogisticRegression()
log_reg.fit(X_train, y_train_propagated)
log_reg.score(X_test, y_test)
```

We got a reasonable accuracy boost, but nothing absolutely astounding - up to 93.1%. The problem is that we propagated each representative instance's label to all the instances in the same cluster, including the instances located close to the cluster boundaries, which are more likely to be mislabeled. Let's see what happens if we only propagate the labels to the 20% of the instances that are closest to the centroids:

```
percentile_closest = 20

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Now let's train the model again on this partially propagated dataset:

```
log_reg = LogisticRegression()
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
log_reg.score(X_test, y_test)
```

Nice! With just 50 labeled instances (only 5 examples per class on average!), we got 94.0% accuracy, which is pretty close to the performance of Logistic Regression on the fully labeled digits dataset (which was 96.9%). This good performance is due to the fact that the propagated labels are actually pretty good—their accuracy is very close to 99%, as the following code shows:

```
np.mean(y_train_partially_propagated == y_train[partially_propagated])
```

---

## Active Learning

To continue improving your model and your training set, the next step could be to do a few rounds of active learning, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called ***uncertainty sampling***. Here is how it works:

1. The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain (i.e., when its estimated probability is lowest) are given to the expert to be labeled.
3. You iterate this process until the performance improvement stops being worth the labeling effort.

Other strategies include labeling the instances that would result in the largest model change, or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM or a Random Forest).

---

Now, let's take a look at DBSCAN, another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

## 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  $\epsilon$  (epsilon) from it. This region is called the instance's  *$\epsilon$ -neighborhood*.
- If an instance has at least `min_samples` instances in its  $\epsilon$ -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.

This algorithm works well if all the clusters are dense enough and if they are well separated by low-density regions. The DBSCAN class in Scikit-Learn is simple to use, as you might expect. Let's test it on the moons dataset, introduced in Module 5:

```
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)
```

The labels of all the instances are now available in the `labels_` instance variable:

```
dbscan.labels_
```

Notice that some instances have a cluster index equal to `-1`, which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
len(dbscan.core_sample_indices_)
dbscan.core_sample_indices_
dbscan.components_
```

```
np.unique(dbscan.labels_)
```

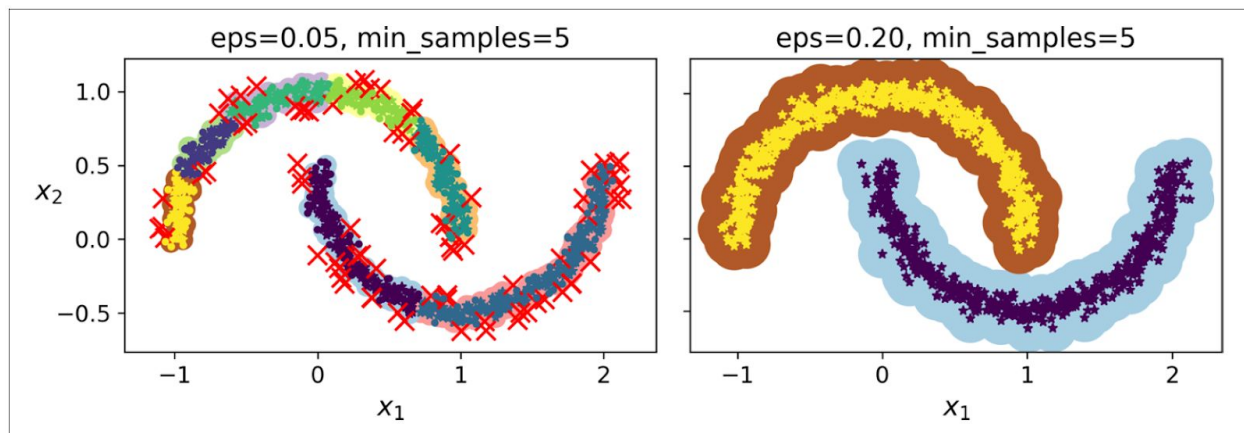
This clustering is represented in the left-hand plot of the following figure, which compares clusters when using  $\epsilon=0.05$  and  $\epsilon=0.20$ .

```
dbscan2 = DBSCAN(eps=0.2)
dbscan2.fit(X)

def plot_dbscan(dbscan, X, size, show_xlabels=True, show_ylabels=True):
    core_mask = np.zeros_like(dbscan.labels_, dtype=bool)
    ...

plt.figure(figsize=(9, 3.2))
```

As you can see, it identified quite a lot of anomalies, plus seven different clusters. How disappointing! Fortunately, if we widen each instance's neighborhood by increasing  $\epsilon$  to 0.2, we get the clustering on the right, which looks perfect. Let's continue with this model.



Somewhat surprisingly, the DBSCAN class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. This implementation decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a `KNeighborsClassifier`:

```
dbscan = dbscan2
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

Now, given a few new instances, we can predict which cluster they most likely belong to and even estimate a probability for each cluster:

```
X_new = np.array([[ -0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)
knn.predict_proba(X_new)
```

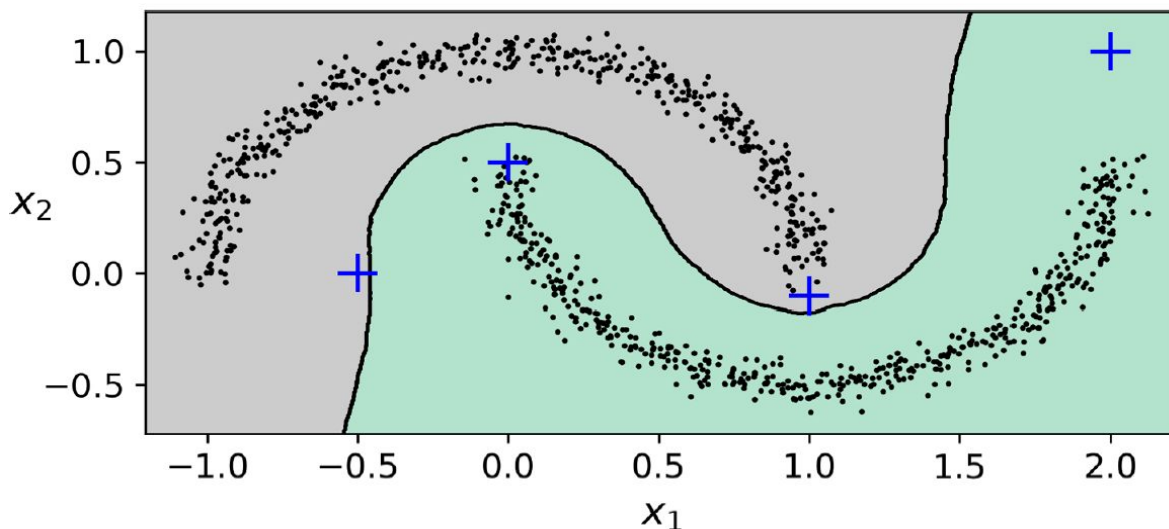
Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies: this choice depends on the final task.

The decision boundary is represented in the following figure (the crosses represent the four instances in  $X_{\text{new}}$ ).

```
plt.figure(figsize=(6, 3))
plot_decision_boundaries(knn, X, show_centroids=False)
plt.scatter(X_new[:, 0], X_new[:, 1], c="b", marker="+", s=200, zorder=10)
save_fig("cluster_classification_plot")
plt.show()
```

Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns the distances and the indices of the  $k$  nearest neighbors in the training set (two matrices, each with  $k$  columns):

```
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[
    [dbscan.core_sample_indices_][y_pred_idx]
]
y_pred[y_dist > 0.2] = -1
y_pred.ravel()
```



In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, however, it can be impossible for it to capture all the clusters properly. Its computational complexity is roughly  $O(m \log m)$ , making it pretty close to linear with regard to the number of instances, but Scikit-Learn's implementation can require up to  $O(m^2)$  memory if `eps` is large.

You may also want to try Hierarchical DBSCAN (HDBSCAN), which is implemented in the `scikit-learn-contrib` project.

**Video: 670\_mod6\_vid12**

## Other Clustering Algorithms

Scikit-Learn implements several more clustering algorithms that you should take a look at. We cannot cover them all in detail here, but here is a brief overview:

### *Spectral clustering*

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces its dimensionality), then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses K-Means.) Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

### *Agglomerative clustering*

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree with a branch for every pair of clusters that merged, you would get a binary tree of clusters, where the leaves are the individual instances. This approach scales very well to large numbers of instances or clusters. It can capture clusters of various shapes, it produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse  $m \times m$  matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.



### *BIRCH*

The BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) algorithm was designed specifically for very large datasets, and it can be faster than batch K-Means, with similar results, as long as the number of features is not too large ( $<20$ ). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the instances in the tree: this approach allows it to use limited memory, while handling huge datasets.

### *Mean-Shift*

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-Shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-Shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the bandwidth), and it relies on local density estimation. But unlike DBSCAN, Mean-Shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is  $O(m^2)$ , so it is not suited for large datasets.

### *Affinity propagation*

This algorithm uses a voting system, where instances vote for similar instances to be their representatives, and once the algorithm converges, each representative and its voters form a cluster. Affinity propagation can detect any number of clusters of different sizes. Unfortunately, this algorithm has a computational complexity of  $O(m^2)$ , so it too is not suited for large datasets.