



K-Means Clustering in Python: A Practical Guide

by Kevin Arvai   **advanced** **data-science** **machine-learning**

Mark as Completed




 Tweet

 Share

 Email

Table of Contents

- [What Is Clustering?](#)
 - [Overview of Clustering Techniques](#)
 - [Partitional Clustering](#)
 - [Hierarchical Clustering](#)
 - [Density-Based Clustering](#)
- [How to Perform K-Means Clustering in Python](#)
 - [Understanding the K-Means Algorithm](#)
 - [Writing Your First K-Means Clustering Code in Python](#)
 - [Choosing the Appropriate Number of Clusters](#)
 - [Evaluating Clustering Performance Using Advanced Techniques](#)
- [How to Build a K-Means Clustering Pipeline in Python](#)
 - [Building a K-Means Clustering Pipeline](#)
 - [Tuning a K-Means Clustering Pipeline](#)
- [Conclusion](#)

 [Remove ads](#)

The **k-means clustering** method is an [unsupervised machine learning](#) technique used to identify clusters of data objects in a dataset. There are many different types of clustering methods, but *k*-means is one of the oldest and most approachable. These traits make implementing *k*-means clustering in Python reasonably straightforward, even for novice programmers and data scientists.

If you're interested in learning how and when to implement *k*-means clustering in Python, then this is the right place. You'll walk through an end-to-end example of *k*-means clustering using Python, from preprocessing the data to evaluating results.

In this tutorial, you'll learn:

- What **k-means clustering** is
- When to use *k*-means clustering to **analyze your data**

Click the link below to download the code you'll use to follow along with the examples in this tutorial and implement your own *k*-means clustering pipeline:

Download the sample code: [Click here to get the code you'll use](#) to learn how to write a *k*-means clustering pipeline in this tutorial.

What Is Clustering?

Clustering is a set of techniques used to partition data into groups, or clusters. **Clusters** are loosely defined as groups of data objects that are more similar to other objects in their cluster than they are to data objects in other clusters. In practice, clustering helps identify two qualities of data:


1. Meaningfulness
2. Usefulness

Meaningful clusters expand domain knowledge. For example, in the medical field, researchers applied clustering to gene expression experiments. The clustering results identified groups of patients who respond differently to medical treatments.

Useful clusters, on the other hand, serve as an intermediate step in a [data pipeline](#). For example, businesses use clustering for customer segmentation. The clustering results segment customers into groups with similar purchase histories, which businesses can then use to create targeted advertising campaigns.

Note: You'll learn about **unsupervised machine learning** techniques in this tutorial. If you're interested in learning more about **supervised machine learning** techniques, then check out [Logistic Regression in Python](#).

There are many other [applications of clustering](#), such as document clustering and social network analysis. These applications are relevant in nearly every industry, making clustering a valuable skill for professionals working with data in any field.

 [Remove ads](#)

Overview of Clustering Techniques

You can perform clustering using many different approaches—so many, in fact, that there are entire categories of clustering algorithms. Each of these categories has its own unique strengths and weaknesses. This means that certain clustering algorithms will result in more natural cluster assignments depending on the input data.

Note: If you're interested in learning about clustering algorithms not mentioned in this section, then check out [A Comprehensive Survey of Clustering Algorithms](#) for an excellent review of popular techniques.

Selecting an appropriate clustering algorithm for your dataset is often difficult due to the number of choices available. Some important factors that affect this decision include the characteristics of the clusters, the features of the dataset, the number of outliers, and the number of data objects.

You'll explore how these factors help determine which approach is most appropriate by looking at three popular categories of clustering algorithms:

1. Partitional clustering
2. Hierarchical clustering
3. Density-based clustering

It's worth reviewing these categories at a high level before jumping right into *k*-means. You'll learn the strengths and weaknesses of each category to provide context for how *k*-means fits into the landscape of clustering algorithms.

Partitional Clustering

Partitional clustering divides data objects into nonoverlapping groups. In other words, no object can be a member of more than one cluster, and every cluster must have at least one object.

These techniques require the user to specify the number of clusters, indicated by the [variable](#) *k*. Many partitional

clustering algorithms work through an iterative process to assign subsets of data points into k clusters. Two examples of partitional clustering algorithms are k -means and k -medoids.

These algorithms are both **nondeterministic**, meaning they could produce different results from two separate runs even if the runs were based on the same input.

Partitional clustering methods have several **strengths**:

- They work well when clusters have a **spherical shape**.
- They're **scalable** with respect to algorithm complexity.

They also have several **weaknesses**:

- They're not well suited for clusters with **complex shapes** and different sizes.
- They break down when used with clusters of different **densities**.

Hierarchical Clustering

Hierarchical clustering determines cluster assignments by building a hierarchy. This is implemented by either a bottom-up or a top-down approach:

- **Agglomerative clustering** is the bottom-up approach. It merges the two points that are the most similar until all points have been merged into a single cluster.
- **Divisive clustering** is the top-down approach. It starts with all points as one cluster and splits the least similar clusters at each step until only single data points remain.

These methods produce a tree-based hierarchy of points called a **dendrogram**. Similar to partitional clustering, in hierarchical clustering the number of clusters (k) is often predetermined by the user. Clusters are assigned by cutting the dendrogram at a specified depth that results in k groups of smaller dendrograms.


Unlike many partitional clustering techniques, hierarchical clustering is a **deterministic** process, meaning cluster assignments won't change when you run an algorithm twice on the same input data.

The **strengths** of hierarchical clustering methods include the following:

- They often reveal the finer details about the **relationships** between data objects.
- They provide an **interpretable dendrogram**.

The **weaknesses** of hierarchical clustering methods include the following:

- They're **computationally expensive** with respect to algorithm complexity.
- They're sensitive to **noise** and **outliers**.

 [Remove ads](#)

Density-Based Clustering

Density-based clustering determines cluster assignments based on the density of data points in a region. Clusters are assigned where there are high densities of data points separated by low-density regions.

Unlike the other clustering categories, this approach doesn't require the user to specify the number of clusters. Instead, there is a distance-based parameter that acts as a tunable threshold. This threshold determines how close points must be to be considered a cluster member.

Examples of density-based clustering algorithms include Density-Based Spatial Clustering of Applications with Noise, or **DBSCAN**, and Ordering Points To Identify the Clustering Structure, or **OPTICS**.

The **strengths** of density-based clustering methods include the following:

- They excel at identifying clusters of **nonspherical shapes**.
- They're resistant to **outliers**.

The **weaknesses** of density-based clustering methods include the following:

- They aren't well suited for **clustering in high-dimensional spaces**.

- They have trouble identifying clusters of **varying densities**.

How to Perform K-Means Clustering in Python

In this section, you'll take a step-by-step tour of the conventional version of the k -means algorithm. Understanding the details of the algorithm is a fundamental step in the process of writing your k -means clustering pipeline in Python. What you learn in this section will help you decide if k -means is the right choice to solve your clustering problem.

Understanding the K-Means Algorithm

Conventional k -means requires only a few steps. The first step is to randomly select k centroids, where k is equal to the number of clusters you choose. **Centroids** are data points representing the center of a cluster.

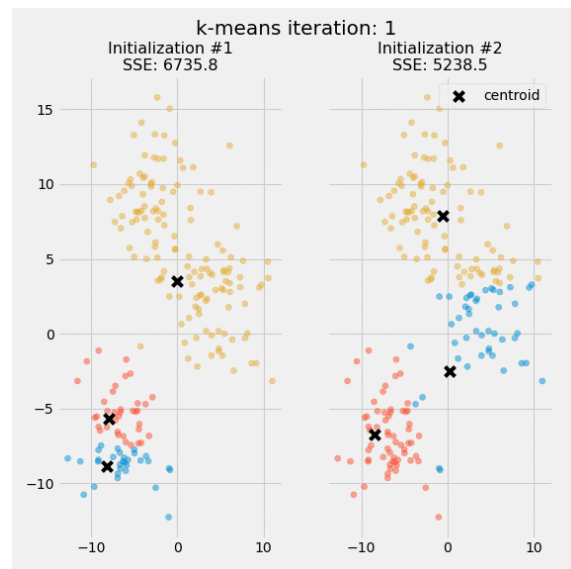
The main element of the algorithm works by a two-step process called **expectation-maximization**. The **expectation** step assigns each data point to its nearest centroid. Then, the **maximization** step computes the mean of all the points for each cluster and sets the new centroid. Here's what the conventional version of the k -means algorithm looks like:

Algorithm 1 k -means algorithm

- 1: Specify the number k of clusters to assign.
 - 2: Randomly initialize k centroids.
 - 3: **repeat**
 - 4: **expectation:** Assign each point to its closest centroid.
 - 5: **maximization:** Compute the new centroid (mean) of each cluster.
 - 6: **until** The centroid positions do not change.
-

The quality of the cluster assignments is determined by computing the [sum of the squared error \(SSE\)](#) after the centroids **converge**, or match the previous iteration's assignment. The SSE is defined as the sum of the squared Euclidean distances of each point to its closest centroid. Since this is a measure of error, the objective of k -means is to try to minimize this value.

The figure below shows the centroids and SSE updating through the first five iterations from two different runs of the k -means algorithm on the same dataset:



The purpose of this figure is to show that the initialization of the centroids is an important step. It also highlights the use of SSE as a measure of clustering performance. After choosing a number of clusters and the initial centroids, the expectation-maximization step is repeated until the centroid positions reach convergence and are unchanged.

The random initialization step causes the k -means algorithm to be **nondeterministic**, meaning that cluster assignments will vary if you run the same algorithm twice on the same dataset. Researchers commonly run several initializations of the entire k -means algorithm and choose the cluster assignments from the initialization with the lowest SSE.

Writing Your First K-Means Clustering Code in Python

Thankfully, there's a robust implementation of *k*-means clustering in Python from the popular machine learning package [scikit-learn](#). You'll learn how to write a practical implementation of the *k*-means algorithm using the [scikit-learn version of the algorithm](#).

Note: If you're interested in gaining a deeper understanding of how to write your own *k*-means algorithm in Python, then check out the [Python Data Science Handbook](#).

The code in this tutorial requires some popular external [Python packages](#) and assumes that you've installed Python with Anaconda. For more information on setting up your Python environment for machine learning in Windows, read through [Setting Up Python for Machine Learning on Windows](#).

Otherwise, you can begin by installing the required packages:

Shell

```
(base) $ conda install matplotlib numpy pandas seaborn scikit-learn ipython
(base) $ conda install -c conda-forge kneed
```

The code is presented so that you can follow along in an ipython console or Jupyter Notebook. Click the prompt (>>>) at the top right of each code block to see the code formatted for copy-paste. You can also download the source code used in this article by clicking on the link below:

Download the sample code: [Click here to get the code you'll use](#) to learn how to write a k-means clustering pipeline in this tutorial.

This step will import the modules needed for all the code in this section:

Python

>>>

```
In [1]: import matplotlib.pyplot as plt
...: from kneed import KneeLocator
...: from sklearn.datasets import make_blobs
...: from sklearn.cluster import KMeans
...: from sklearn.metrics import silhouette_score
...: from sklearn.preprocessing import StandardScaler
```

You can generate the data from the above GIF using `make_blobs()`, a convenience function in scikit-learn used to generate synthetic clusters. `make_blobs()` uses these parameters:

- **n_samples** is the total number of samples to generate.
- **centers** is the number of centers to generate.
- **cluster_std** is the standard deviation.

`make_blobs()` returns a tuple of two values:

1. A two-dimensional NumPy array with the **x- and y-values** for each of the samples
2. A one-dimensional NumPy array containing the **cluster labels** for each sample

Note: Many scikit-learn algorithms rely heavily on NumPy in their implementations. If you want to learn more about NumPy arrays, check out [Look Ma, No For-Loops: Array Programming With NumPy](#).

Generate the synthetic data and labels:

Python

>>>

```
In [2]: features, true_labels = make_blobs(
...:     n_samples=200,
...:     centers=3,
...:     cluster_std=2.75,
...:     random_state=42
...: )
```

Nondeterministic machine learning algorithms like *k*-means are difficult to reproduce. The `random_state` parameter

is set to an integer value so you can follow the data presented in the tutorial. In practice, it's best to leave `random_state` as the default value, `None`.

Here's a look at the first five elements for each of the variables returned by `make_blobs()`:

Python

>>>

```
In [3]: features[:5]
Out[3]:
array([[ 9.77075874,  3.27621022],
       [-9.71349666, 11.27451802],
       [-6.91330582, -9.34755911],
       [-10.86185913, -10.75063497],
       [-8.50038027, -4.54370383]])

In [4]: true_labels[:5]
Out[4]: array([1, 0, 2, 2, 2])
```

Data sets usually contain numerical features that have been measured in different units, such as height (in inches) and weight (in pounds). A machine learning algorithm would consider weight more important than height only because the values for weight are larger and have higher variability from person to person.

Machine learning algorithms need to consider all features on an even playing field. That means the values for all features must be transformed to the same scale.

The process of transforming numerical features to use the same scale is known as **feature scaling**. It's an important data **preprocessing** step for most distance-based machine learning algorithms because it can have a significant impact on the performance of your algorithm.

There are several approaches to implementing feature scaling. A great way to determine which technique is appropriate for your dataset is to read scikit-learn's [preprocessing documentation](#).

In this example, you'll use the `StandardScaler` class. This class implements a type of feature scaling called **standardization**. Standardization scales, or shifts, the values for each numerical feature in your dataset so that the features have a mean of 0 and standard deviation of 1:

Python

>>>

```
In [5]: scaler = StandardScaler()
...: scaled_features = scaler.fit_transform(features)
```

Take a look at how the values have been scaled in `scaled_features`:

Python

>>>

```
In [6]: scaled_features[:5]
Out[6]:
array([[ 2.13082109,  0.25604351],
       [-1.52698523,  1.41036744],
       [-1.00130152, -1.56583175],
       [-1.74256891, -1.76832509],
       [-1.29924521, -0.87253446]])
```

Now the data are ready to be clustered. The `KMeans` estimator class in scikit-learn is where you set the algorithm parameters before fitting the estimator to the data. The scikit-learn implementation is flexible, providing several parameters that can be tuned.

Here are the parameters used in this example:

- **init** controls the initialization technique. The standard version of the *k*-means algorithm is implemented by setting `init` to "random". Setting this to "k-means++" employs an advanced trick to speed up convergence, which you'll use later.
- **n_clusters** sets *k* for the clustering step. This is the most important parameter for *k*-means.
- **n_init** sets the number of initializations to perform. This is important because two runs can converge on different cluster assignments. The default behavior for the scikit-learn algorithm is to perform ten *k*-means runs and return the results of the one with the lowest SSE.

- **max_iter** sets the number of maximum iterations for each initialization of the *k*-means algorithm.

Instantiate the `KMeans` class with the following arguments:

Python

>>>

```
In [7]: kmeans = KMeans(  
...:     init="random",  
...:     n_clusters=3,  
...:     n_init=10,  
...:     max_iter=300,  
...:     random_state=42  
...: )
```

The parameter names match the language that was used to describe the *k*-means algorithm earlier in the tutorial. Now that the *k*-means class is ready, the next step is to fit it to the data in `scaled_features`. This will perform ten runs of the *k*-means algorithm on your data with a maximum of 300 iterations per run:

Python

>>>

```
In [8]: kmeans.fit(scaled_features)  
Out[8]:  
KMeans(init='random', n_clusters=3, random_state=42)
```

Statistics from the initialization run with the lowest SSE are available as attributes of `kmeans` after calling `.fit()`:

Python

>>>

```
In [9]: # The lowest SSE value  
...: kmeans.inertia_  
Out[9]: 74.57960106819854  
  
In [10]: # Final locations of the centroid  
...: kmeans.cluster_centers_  
Out[10]:  
array([[ 1.19539276,  0.13158148],  
       [-0.25813925,  1.05589975],  
       [-0.91941183, -1.18551732]])  
  
In [11]: # The number of iterations required to converge  
...: kmeans.n_iter_  
Out[11]: 6
```

Finally, the cluster assignments are stored as a one-dimensional NumPy array in `kmeans.labels_`. Here's a look at the first five predicted labels:

Python

>>>

```
In [12]: kmeans.labels_[:5]  
Out[12]: array([0, 1, 2, 2], dtype=int32)
```

Note that the order of the cluster labels for the first two data objects was flipped. The order was `[1, 0]` in `true_labels` but `[0, 1]` in `kmeans.labels_` even though those data objects are still members of their original clusters in `kmeans.labels_`.

This behavior is normal, as the ordering of cluster labels is dependent on the initialization. Cluster 0 from the first run could be labeled cluster 1 in the second run and vice versa. This doesn't affect clustering evaluation metrics.

 [Remove ads](#)

Choosing the Appropriate Number of Clusters

In this section, you'll look at two methods that are commonly used to evaluate the appropriate number of clusters:

1. The **elbow method**
2. The **silhouette coefficient**

These are often used as complementary evaluation techniques rather than one being preferred over the other. To perform the **elbow method**, run several *k*-means, increment *k* with each iteration, and record the SSE:

Python

>>>

```
In [13]: kmeans_kwargs = {
...:     "init": "random",
...:     "n_init": 10,
...:     "max_iter": 300,
...:     "random_state": 42,
...: }
...:
...: # A list holds the SSE values for each k
...: sse = []
...: for k in range(1, 11):
...:     kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
...:     kmeans.fit(scaled_features)
...:     sse.append(kmeans.inertia_)
```

The previous code block made use of Python's dictionary unpacking operator (**). To learn more about this powerful Python operator, check out [How to Iterate Through a Dictionary in Python](#).

When you plot SSE as a function of the number of clusters, notice that SSE continues to decrease as you increase k. As more centroids are added, the distance from each point to its closest centroid will decrease.

There's a sweet spot where the SSE curve starts to bend known as the **elbow point**. The x-value of this point is thought to be a reasonable trade-off between error and number of clusters. In this example, the elbow is located at x=3:

Python

>>>

```
In [14]: plt.style.use("fivethirtyeight")
...: plt.plot(range(1, 11), sse)
...: plt.xticks(range(1, 11))
...: plt.xlabel("Number of Clusters")
...: plt.ylabel("SSE")
...: plt.show()
```

The above code produces the following plot:



Determining the elbow point in the SSE curve isn't always straightforward. If you're having trouble choosing the elbow point of the curve, then you could use a Python package, [kneed](#), to identify the elbow point programmatically:

Python

>>>

```
In [15]: kl = KneLocator(
...:     range(1, 11), sse, curve="convex", direction="decreasing"
...: )

In [16]: kl.elbow
Out[16]: 3
```

The **silhouette coefficient** is a measure of cluster cohesion and separation. It quantifies how well a data point fits into its assigned cluster based on two factors:

1. How close the data point is to other points in the cluster
2. How far away the data point is from points in other clusters

Silhouette coefficient values range between -1 and 1. Larger numbers indicate that samples are closer to their clusters than they are to other clusters.

In the scikit-learn [implementation of the silhouette coefficient](#), the average silhouette coefficient of all the samples is summarized into one score. The `silhouette_score()` function needs a minimum of two clusters, or it will raise an exception.

Loop through values of k again. This time, instead of computing SSE, compute the silhouette coefficient:

Python

>>>

```
In [17]: # A list holds the silhouette coefficients for each k
...: silhouette_coefficients = []
...:
...: # Notice you start at 2 clusters for silhouette coefficient
...: for k in range(2, 11):
...:     kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
...:     kmeans.fit(scaled_features)
...:     score = silhouette_score(scaled_features, kmeans.labels_)
...:     silhouette_coefficients.append(score)
```

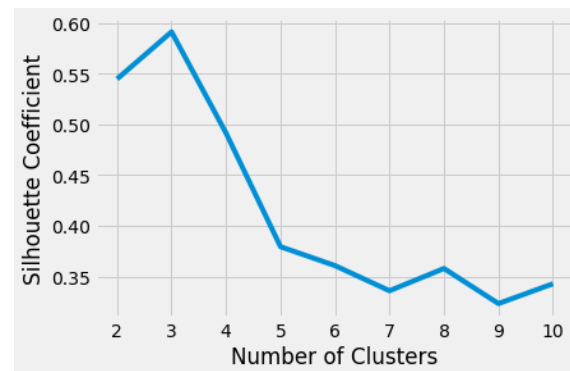
Plotting the average silhouette scores for each k shows that the best choice for k is 3 since it has the maximum score:

Python

>>>

```
In [18]: plt.style.use("fivethirtyeight")
...: plt.plot(range(2, 11), silhouette_coefficients)
...: plt.xticks(range(2, 11))
...: plt.xlabel("Number of Clusters")
...: plt.ylabel("Silhouette Coefficient")
...: plt.show()
```

The above code produces the following plot:



Ultimately, your decision on the number of clusters to use should be guided by a combination of domain knowledge and clustering evaluation metrics.

[Remove ads](#)

Evaluating Clustering Performance Using Advanced Techniques

The elbow method and silhouette coefficient evaluate clustering performance without the use of **ground truth labels**. Ground truth labels categorize data points into groups based on assignment by a human or an existing algorithm. These types of metrics do their best to suggest the correct number of clusters but can be deceiving when used without context.

Note: In practice, it's rare to encounter datasets that have ground truth labels.

When comparing k -means against a density-based approach on nonspherical clusters, the results from the elbow method and silhouette coefficient rarely match human intuition. This scenario highlights why advanced clustering evaluation techniques are necessary. To visualize an example, import these additional modules:

Python

>>>

```
In [19]: from sklearn.cluster import DBSCAN
...: from sklearn.datasets import make_moons
...: from sklearn.metrics import adjusted_rand_score
```

This time, use `make_moons()` to generate synthetic data in the shape of crescents:

Python

>>>

```
In [20]: features, true_labels = make_moons(
...:     n_samples=250, noise=0.05, random_state=42
...: )
...: scaled_features = scaler.fit_transform(features)
```

Fit both a *k*-means and a DBSCAN algorithm to the new data and visually assess the performance by plotting the cluster assignments with [Matplotlib](#):

Python

>>>

```
In [21]: # Instantiate k-means and dbscan algorithms
...: kmeans = KMeans(n_clusters=2)
...: dbscan = DBSCAN(eps=0.3)
...:
...: # Fit the algorithms to the features
...: kmeans.fit(scaled_features)
...: dbscan.fit(scaled_features)
...:
...: # Compute the silhouette scores for each algorithm
...: kmeans_silhouette = silhouette_score(
...:     scaled_features, kmeans.labels_
...: ).round(2)
...: dbscan_silhouette = silhouette_score(
...:     scaled_features, dbscan.labels_
...: ).round(2)
```

Print the silhouette coefficient for each of the two algorithms and compare them. A higher silhouette coefficient suggests better clusters, which is misleading in this scenario:

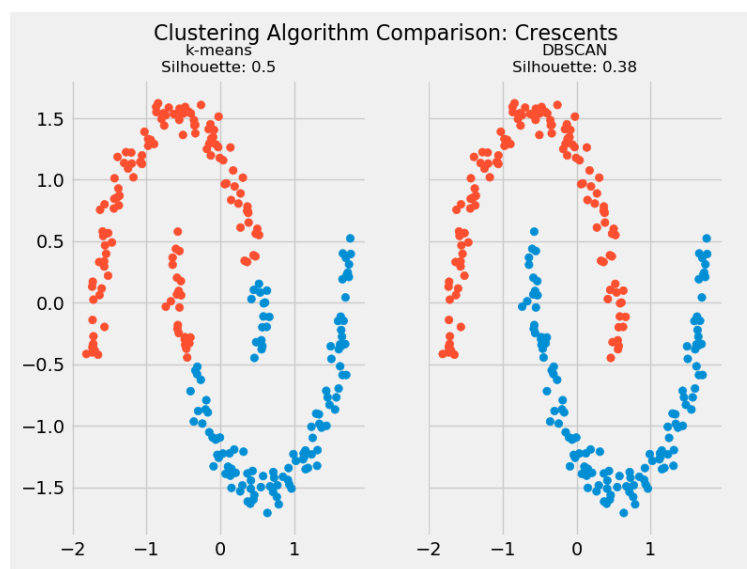
Python

>>>

```
In [22]: kmeans_silhouette
Out[22]: 0.5

In [23]: dbscan_silhouette
Out[23]: 0.38
```

The silhouette coefficient is higher for the *k*-means algorithm. The DBSCAN algorithm appears to find more natural clusters according to the shape of the data:



This suggests that you need a better method to compare the performance of these two clustering algorithms.

If you're interested, you can find the code for the above plot by expanding the box below.

Plot Crescents

Show/Hide

Since the ground truth labels are known, it's possible to use a clustering metric that considers labels in its evaluation. You can use the [scikit-learn implementation](#) of a common metric called the **adjusted rand index (ARI)**. Unlike the silhouette coefficient, the ARI uses true cluster assignments to measure the similarity between true and predicted labels.

Compare the clustering results of DBSCAN and *k*-means using ARI as the performance metric:

Python

>>>

```
In [25]: ari_kmeans = adjusted_rand_score(true_labels, kmeans.labels_)
...: ari_dbscan = adjusted_rand_score(true_labels, dbscan.labels_)


In [26]: round(ari_kmeans, 2)
Out[26]: 0.47

In [27]: round(ari_dbscan, 2)
Out[27]: 1.0
```

The ARI output values range between -1 and 1 . A score close to 0.0 indicates random assignments, and a score close to 1 indicates perfectly labeled clusters.

Based on the above output, you can see that the silhouette coefficient was misleading. ARI shows that DBSCAN is the best choice for the synthetic crescents example as compared to *k*-means.

There are several metrics that evaluate the quality of clustering algorithms. Reading through [the implementations in scikit-learn](#) will help you select an appropriate clustering evaluation metric.

 [Remove ads](#)

How to Build a K-Means Clustering Pipeline in Python

Now that you have a basic understanding of *k*-means clustering in Python, it's time to perform *k*-means clustering on a real-world dataset. These data contain gene expression values from a manuscript authored by [The Cancer Genome Atlas](#) (TCGA) Pan-Cancer analysis project investigators.

There are 881 samples (rows) representing five distinct cancer subtypes. Each sample has gene expression values for 20,531 genes (columns). The [dataset](#) is available from the [UC Irvine Machine Learning Repository](#), but you can use the Python code below to obtain the data programmatically.

To follow along with the examples below, you can download the source code by clicking on the following link:

Download the sample code: [Click here to get the code you'll use](#) to learn how to write a *k*-means clustering pipeline in this tutorial.

In this section, you'll build a robust *k*-means clustering pipeline. Since you'll perform multiple transformations of the original input data, your pipeline will also serve as a practical clustering framework.

Building a K-Means Clustering Pipeline

Assuming you want to start with a fresh [namespace](#), import all the modules needed to build and evaluate the pipeline, including [pandas](#) and [seaborn](#) for more advanced visualizations: