Learning K-Means Clustering with Python: A Step-by-Step Tutorial

Authored by

Mohammed looti

October 27, 2025

Introduction to K-Means Clustering

Clustering algorithms form a foundational pillar of unsupervised machine learning, enabling data scientists to discover inherent groupings within datasets without relying on labeled outcomes. Among these techniques, **K-means clustering** stands out as perhaps the most widely recognized and frequently implemented method due to its simplicity and computational efficiency. It provides an effective means of partitioning *n* observations into *K* distinct clusters.

The core objective of the K-means algorithm is straightforward: to define *K* clusters such that the total within-cluster variation is minimized. This means that observations assigned to the same cluster should exhibit high similarity to one another, typically measured by their spatial proximity in the feature space. Conversely, observations belonging to different clusters should be distinctly dissimilar.

Choosing the appropriate value for *K*--the number of clusters--is often the most critical and subjective step in the process. While the algorithm handles the assignment of points iteratively, the initial selection of *K* dictates the granularity of the resulting structure. Throughout this guide, we will walk through a practical example in Python using the powerful **scikit-learn** library, demonstrating how to apply K-means to real-world data, from preparation to final interpretation.

The K-Means Algorithm Explained

Understanding the iterative nature of K-means is essential for effective implementation. The algorithm operates by repeatedly optimizing the cluster assignments until convergence is achieved--that is, until the cluster memberships cease to change significantly. This convergence ensures that the resulting clusters are as compact and separated as possible given the specified value of *K*.

In practice, the K-means procedure follows a precise sequence of steps designed to minimize the squared error between the data points and the cluster centroid. A centroid is simply the arithmetic mean (average) of all the points belonging to that cluster, representing the cluster's center point in the feature space.

We perform **K-means clustering** using the following standardized steps:

Initialization: Choose the Number of Clusters (K). The process begins with the determination of *K*, the desired number of clusters. Since this choice is not always obvious, empirical methods like the Elbow Method (discussed later) are often used in conjunction with domain expertise to select an optimal value. This step fundamentally defines the segmentation goal.

Random Assignment of Initial Centroids. The algorithm randomly selects *K* initial data points

from the dataset to serve as the starting centroids. Alternatively, it can randomly assign each observation to one of the *K* clusters initially. The starting position of these centroids can significantly influence the final clustering result, which is why the algorithm is often run multiple times with different initializations.

Iterative Refinement (E-step and M-step). The algorithm proceeds through alternating steps until cluster assignments stabilize:

E-step (Expectation): Each observation in the dataset is assigned to the cluster whose centroid is nearest. The determination of "nearest" is typically calculated using the squared Euclidean distance, which measures the straight-line distance between the observation and each centroid.

M-step (Maximization): Once all observations are assigned, the centroids for all *K* clusters are recalculated. The new centroid for a cluster is determined by computing the mean of all feature values for the observations currently belonging to that cluster.

Convergence. The E-step and M-step are repeated until the cluster assignments no longer change, or until a maximum number of iterations is reached. At this point, the algorithm is considered converged, and the final cluster assignments are returned.

Setting Up the Python Environment and Data

To perform **K-means clustering** in Python, we rely on established libraries for data handling and machine learning execution. Specifically, **pandas** is crucial for efficient data manipulation, **numpy** supports numerical operations, and **matplotlib** allows for visualization, particularly when determining the optimal *K*. The core functionality, including the `KMeans` function and data preparation tools, is sourced from the powerful **scikit-learn (sklearn)** library.

Our first task is to import these necessary modules into the Python environment. This ensures that all required functions are accessible for the subsequent steps of data preparation and model fitting.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
```

Next, we construct the dataset that we will analyze. For this example, we generate a synthetic DataFrame containing statistics for twenty different basketball players. Each player is characterized by three key metrics that quantify their performance: **points** scored, **assists** distributed, and **rebounds** secured. Our ultimate goal is to use K-means to group players who

exhibit similar statistical profiles across these three dimensions.

#create DataFrame
df = pd.DataFrame({'points': ,
'assists': ,
'rebounds': })

#view first five rows of DataFrame
print(df.head())

points assists rebounds
0 18.0 3.0 15
1 NaN 3.0 14
2 19.0 4.0 14
3 14.0 5.0 10
4 14.0 4.0 8

Data Preprocessing and Normalization

Before applying the K-means algorithm, the data must be rigorously cleaned and prepared. Clustering algorithms are highly sensitive to missing values and the scale of the input features. If data preparation is skipped, the results can be unreliable or heavily skewed toward features with larger numerical ranges.

The first critical step in preprocessing is handling missing data, denoted by `NaN` values in our DataFrame. Since K-means requires complete data for distance calculations, we must remove or impute these missing entries. For simplicity in this tutorial, we opt to remove any rows containing `NaN` values using the **dropna()** method.

The second, and perhaps most crucial, preprocessing step for distance-based algorithms like K-means is **feature scaling**. If variables are measured on vastly different scales (e.g., points ranging from 10 to 35, while assists range from 3 to 15), the variables with larger ranges will exert a disproportionate influence on the Euclidean distance calculation. To ensure every feature contributes equally to the clustering process, we employ standardization.

Standardization transforms the data such that each feature has a mean of 0 and a standard deviation of 1. This process is often referred to as Z-score normalization. We achieve this using scikit-learn's **StandardScaler()** function, which is imported in Step 1. The following code executes both the cleaning and the scaling operations:

#drop rows with NA values in any columns

df = df.dropna()

#create scaled DataFrame where each variable has mean of 0 and standard dev of 1
scaled_df = StandardScaler().fit_transform(df)

#view first five rows of scaled DataFrame
print(scaled_df)

]

Determining the Optimal Number of Clusters (K)

The success of K-means hinges on selecting an appropriate value for *K*, the number of clusters. While the `KMeans` function in `sklearn.cluster` is central to the implementation, it requires the `n_clusters` parameter to be explicitly set. The basic structure of the function, along with its key arguments, is detailed below:

KMeans(init='random', n_clusters=8, n_init=10, random_state=None)

init: Controls the initialization technique for the centroids (e.g., 'random' or 'k-means++' for smarter initialization).

n_clusters: The critical parameter specifying the number of clusters to form.

n_init: The number of times the K-means algorithm will be run with different centroid seeds. The final result will be the best output in terms of lowest **Sum of Squared Errors (SSE)**.

random_state: An integer used to ensure reproducibility across runs.

Since we do not know the optimal *K* beforehand, we employ the **Elbow Method**. The Elbow Method involves running K-means for a range of *K* values and calculating the SSE for each model. SSE measures the sum of the squared distances between each point and the centroid of its assigned cluster. As *K* increases, the SSE naturally decreases (since more clusters mean points are closer to their respective centers).

We plot *K* (Number of Clusters) against the resulting SSE. The optimal number of clusters is typically found at the "elbow"--the point on the curve where the rate of decrease in SSE sharply slows down or "bends." This bend signifies that adding further clusters provides diminishing returns in terms of reducing within-cluster variance. The following Python code iterates through *K* values from 1 to 10, calculates the SSE for each, and visualizes the results:

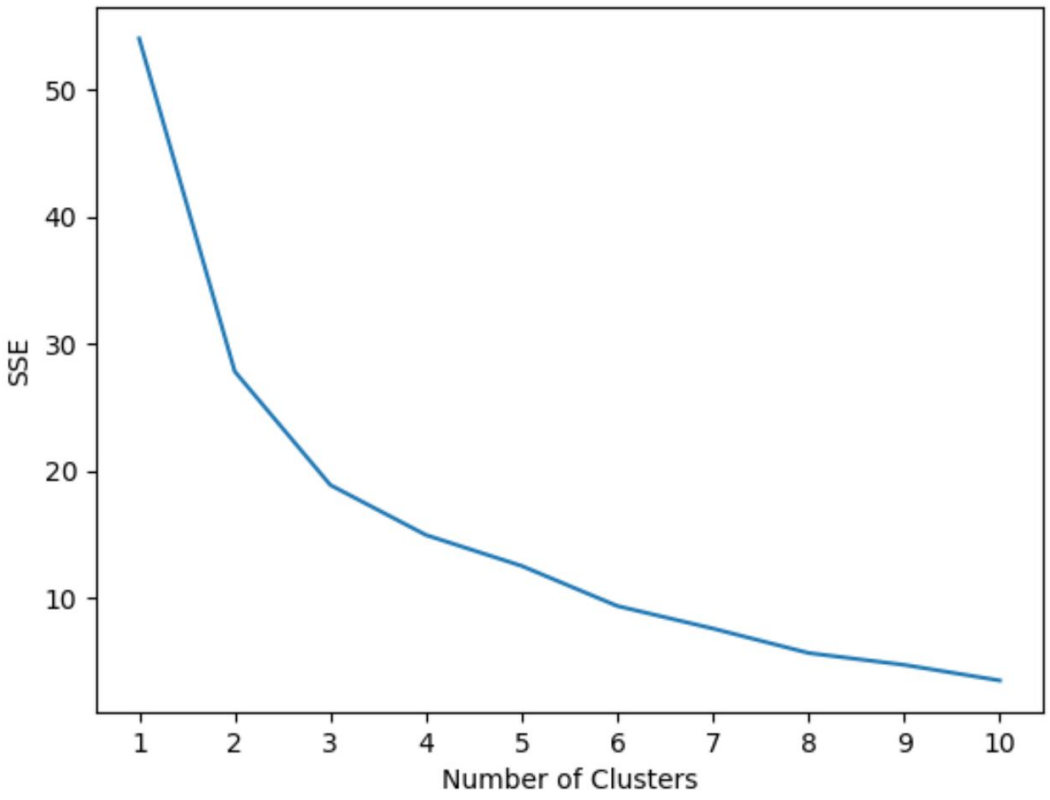
#initialize kmeans parameters

```
kmeans_kwargs = {
"init": "random",
"n_init": 10,
"random_state": 1,
}

#create list to hold SSE values for each k
sse =
for k in range(1, 11):
kmeans = KMeans(n_clusters=k, **kmeans_kwargs)
kmeans.fit(scaled_df)
sse.append(kmeans.inertia_)

#visualize results
plt.plot(range(1, 11), sse)
plt.xticks(range(1, 11))
plt.xlabel("Number of Clusters")
plt.ylabel("SSE")
plt.show()
```

The resulting plot below clearly illustrates the trade-off between the number of clusters and the total within-cluster variance.



Upon inspection of the graph, a distinct "elbow" or significant leveling off occurs when *K* is equal to **3 clusters**. While the SSE continues to decrease past 3, the marginal benefit of adding a fourth or fifth cluster is minimal compared to the reduction achieved by moving from 2 to 3 clusters. Based on this visual evidence, we select *K*=3 as the optimal number for our clustering model.

Executing and Interpreting the K-Means Model

With the optimal number of clusters determined, we can now fit the final K-means model to our scaled data. We initialize the `KMeans` class using `n_clusters=3` and fit it to the `scaled_df`, ensuring consistent initialization parameters (`init='random'`, `n_init=10`, `random_state=1`).

The code below executes the final clustering process:

```
#instantiate the k-means class, using optimal number of clusters
kmeans = KMeans(init="random", n_clusters=3, n_init=10, random_state=1)

#fit k-means algorithm to data
kmeans.fit(scaled_df)

#view cluster assignments for each observation
kmeans.labels_
```

array()

The output array, generated by accessing the `kmeans.labels_` attribute, represents the cluster assignment (0, 1, or 2) for each observation in the dataset. To make these results meaningful and link them back to the original player statistics, we must append these cluster labels to our initial, unscaled DataFrame.

By adding a dedicated `cluster` column, we can easily analyze the characteristics of players within each group. This final step allows us to interpret the physical meaning of the clusters--for instance, identifying Cluster 0 as "High Volume Scorers" or Cluster 1 as "Rebound Specialists."

#append cluster assingments to original DataFrame
df = kmeans.labels_

#view updated DataFrame
print(df)

points assists rebounds cluster
0 18.0 3.0 15 1
2 19.0 4.0 14 1
3 14.0 5.0 10 1
4 14.0 4.0 8 1
5 11.0 7.0 14 1
6 20.0 8.0 13 1
7 28.0 7.0 9 2
8 30.0 6.0 5 2
9 31.0 9.0 4 0
10 35.0 12.0 11 0
11 33.0 14.0 6 0
13 25.0 9.0 5 0
14 25.0 4.0 3 2
15 27.0 3.0 8 2
16 29.0 4.0 12 2
17 30.0 12.0 7 0
18 19.0 15.0 6 0
19 23.0 11.0 5 0

The resulting DataFrame now clearly organizes the basketball players into three distinct groups based on the similarity of their **points**, **assists**, and **rebounds** statistics. For instance, players assigned to Cluster 1 tend to have high rebound counts, while players in Cluster 0 generally exhibit

high points and assists. This clustering provides immediate, actionable insights into the natural segmentation of the players based on their performance metrics.

Conclusion and Further Resources

K-means clustering remains a fundamental and highly effective tool for exploratory data analysis and segmentation tasks in machine learning. By following a systematic approach--from data scaling and normalization using **StandardScaler** to optimizing the cluster count via the Elbow Method--we successfully partitioned a dataset of basketball players into three meaningful performance groups using Python and the scikit-learn library. The resulting clusters provide a powerful summary of the data structure, facilitating deeper analysis and informed decision-making based on inherent data patterns.

For users looking to expand their knowledge of Python data science and machine learning tasks, the following resources are recommended:

Official documentation for the **KMeans** function from **sklearn**.

Tutorials explaining how to handle different types of missing data (imputation techniques).

Guides on alternative clustering methods, such as DBSCAN or Hierarchical Clustering.