How to Implement K-Means Clustering in Python

Authored by

stats writer

November 24, 2025

K-means clustering is perhaps the most widely recognized algorithm within the field of unsupervised machine learning. Its primary function is to partition a dataset into a predefined number of groups, denoted as **K**, where data points within the same group are significantly more similar to one another than to those in other groups. This method is incredibly powerful for discovering inherent groupings in unlabeled data, making it essential for tasks like market segmentation, anomaly detection, and document classification.

Implementing a robust K-means model requires a methodical approach in Python, encompassing crucial steps like data preprocessing, determining the optimal number of clusters, and leveraging specialized libraries such as Scikit-learn. This tutorial will guide you through the process using a real-world example, ensuring clean data handling and accurate model fitting using the powerful `KMeans` class.

The Iterative Process of K-Means Clustering

The K-means algorithm operates through an iterative refinement process aimed at minimizing the variance within each cluster. Understanding this underlying mechanism is crucial before diving into the implementation. The process generally follows three core stages: initialization, calculation of centroids, and reassignment of data points.

This iterative procedure continues until convergence is achieved--meaning the cluster assignments no longer change, or a maximum number of iterations is reached.

Determine the Number of Clusters (K): The algorithm requires the user to specify **K**, the desired number of clusters. This is often the most subjective step. In practical applications, the optimal value for **K** is often determined by experimenting with different values and evaluating the results using statistical measures like the Elbow Method, which we will explore later.

Initialization of Centroids: Once **K** is chosen, the algorithm randomly selects **K** data points from the dataset to serve as the initial cluster centers, or **centroids**. Modern libraries often use smarter initialization techniques (like k-means++) to speed up convergence and avoid poor local minima, but the core concept remains the same.

Assignment and Update Loop: This is the heart of the algorithm, executed repeatedly until stability is reached:

Assignment Step: Every observation in the dataset is assigned to the cluster whose centroid is nearest. This proximity is measured using a distance metric, typically the Euclidean distance, which calculates the straight-line distance between the data point and the centroid.
Update Step: After all points have been assigned, the **centroid** of each cluster is recalculated. The new centroid is simply the vector of feature means (the mean of all features) for all

observations currently belonging to that specific cluster.

We will now transition to implementing this process in Python using a sample dataset related to athlete statistics, utilizing the highly efficient clustering tools provided by the Scikit-learn library.

Step 1: Importing Required Libraries

The first action in any Python data science project is importing the necessary dependencies. For K-means clustering, we rely on a stack of powerful tools: **Pandas** for efficient data handling and manipulation, **NumPy** for numerical operations, **Matplotlib** for visualization, and the specific modules from Scikit-learn (`KMeans` and `StandardScaler`) that handle the modeling and preprocessing tasks.

Execute the following lines to ensure all modules are available in your environment before proceeding to data creation and cleaning.

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

Step 2: Generating and Inspecting the Dataset

For this demonstration, we will analyze performance data for twenty hypothetical basketball players. The objective of the K-means model will be to group players who exhibit similar statistical profiles across three key metrics.

The chosen features for clustering are:

points (Average Points Scored)
assists (Average Assists per Game)
rebounds (Average Rebounds per Game)

The following Pandas code block constructs the DataFrame, `df`, which contains raw numerical data. Note that we have intentionally included `np.nan` values to simulate missing data, preparing us for the necessary data cleaning phase in the next step.

```
#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

By clustering players based on these composite statistics, we expect the K-means clustering algorithm to naturally identify distinct player archetypes (e.g., scorers, playmakers, rebounders) without being explicitly told what those groups are.

Step 3: Data Preprocessing (Handling Missing Values and Scaling)

Before any clustering algorithm can be effectively applied, the data must be rigorously cleaned and prepared. Machine learning models, including K-means, cannot handle missing values (NaN), and are highly sensitive to the scale and magnitude of the input features.

Our preparation process involves two mandatory steps: first, removing observations with missing data using the Pandas `.dropna()` method. Second, we employ **Standardization**. Standardization rescales the features such that they have a mean of 0 and a standard deviation of 1. This crucial step prevents features with larger numerical ranges (like 'points') from unduly dominating the distance calculations used by K-means, ensuring that all features contribute equally to defining cluster membership.

The StandardScaler utility from Scikit-learn is ideal for this task. It transforms the DataFrame into a NumPy array suitable for model input.

#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)

]

As shown in the output, the scaled data now consists of standardized values, ready for the clustering algorithm to process.

Step 4: Determining the Optimal K Value Using the Elbow Method

A critical challenge in K-means implementation is selecting the appropriate value for **n_clusters**, or **K**. Using the wrong K value can lead to under-segmentation or over-segmentation of the data. We address this uncertainty using the Elbow Method, a heuristic technique that relies on measuring the within-cluster variance.

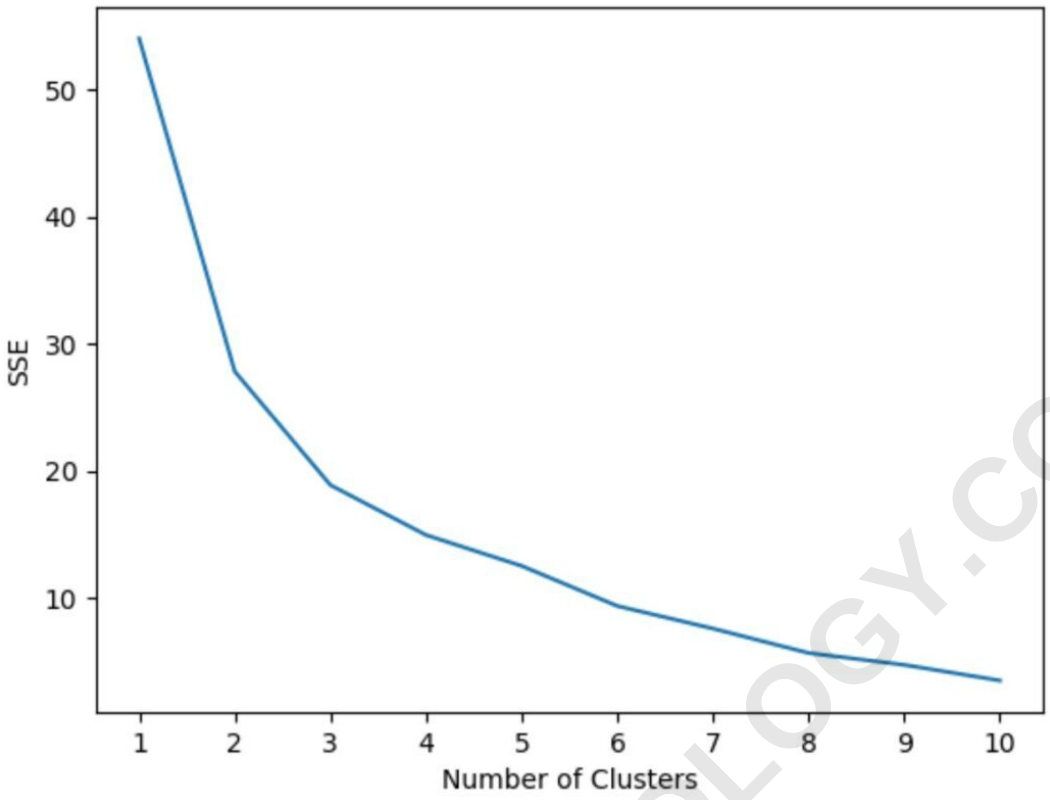
The core metric evaluated by the Elbow Method is the **Sum of Squared Errors (SSE)**, also known as inertia. SSE measures the sum of the squared distances between each data point and the centroid of its assigned cluster. We iterate the K-means algorithm across a range of possible K values (e.g., K=1 through K=10) and record the resulting SSE for each iteration.

The syntax for the `KMeans` function in Scikit-learn allows for fine-tuning parameters, including `n_init` (the number of times the algorithm is run with different centroid seeds, returning the best result) and `random_state` for reproducibility. The following code calculates the SSE for K values ranging from 1 to 10 and then generates the diagnostic plot.

```
#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()
```



In the resulting plot, the optimal K value is visually identified where the marginal gain in reducing the SSE begins to level off, forming an "elbow" shape. Analyzing this graph, a significant bend is visible at K = **3 clusters**. Although this method provides a strong statistical suggestion, remember that in professional data analysis, this result should always be validated against domain knowledge to ensure the clusters are meaningful.

Step 5: Executing K-Means Clustering and Interpreting Results

With the optimal number of clusters determined (K=3), we can now instantiate the `KMeans` model with this parameter and fit it to our standardized data (`scaled_df`). The fitting process calculates the final centroids and assigns a definitive cluster label to every observation based on the minimum distance to these centroids.

We use the same initialization parameters as in the Elbow Method calculation to ensure consistency. Once the model is fitted, the cluster assignments for each player are stored in the `.labels_` attribute of the `KMeans` object.

#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 resulting array provides the cluster index (0, 1, or 2) assigned to each player observation in the order they appeared in the cleaned DataFrame. To make these results actionable and easier to understand in context, it is best practice to merge these labels back into the original, non-scaled DataFrame.

By adding a new column named `cluster` to the DataFrame, we can now analyze the raw statistics alongside the assigned group. This step enables us to characterize and name the clusters (e.g., "High Volume Scorers," "Defensive Specialists," etc.) based on the average features within each group.

```
#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

Observation of the output confirms that players grouped into the same cluster index (0, 1, or 2) exhibit shared characteristics across the **points**, **assists**, and **rebounds** features. For instance, Cluster 1 appears to contain players with low points and assists but very high rebounds, suggesting a defensive/rebounding specialization. Cluster 0, conversely, shows high points and high assists, indicative of primary offensive playmakers.

Conclusion and Next Steps

K-means is a fundamental and efficient algorithm for performing unsupervised machine learning tasks. By systematically applying data preprocessing techniques like standardization and utilizing the Elbow Method to select an optimal **K**, we successfully segmented a dataset of basketball players into three statistically distinct performance groups.

To delve deeper into the implementation details and explore advanced parameters not covered here, readers are encouraged to consult the official documentation for the `KMeans` classes within the Scikit-learn library. Mastering these tools is key to solving complex clustering problems in various domains.

The following tutorials explain how to perform other common tasks in Python: