Clustering in Action: Exploring K-Means with Scikit-Learn

An introduction to clustering and how to implement K-Means efficiently using scikit-learn.

Hernández Pérez Erick Fernando

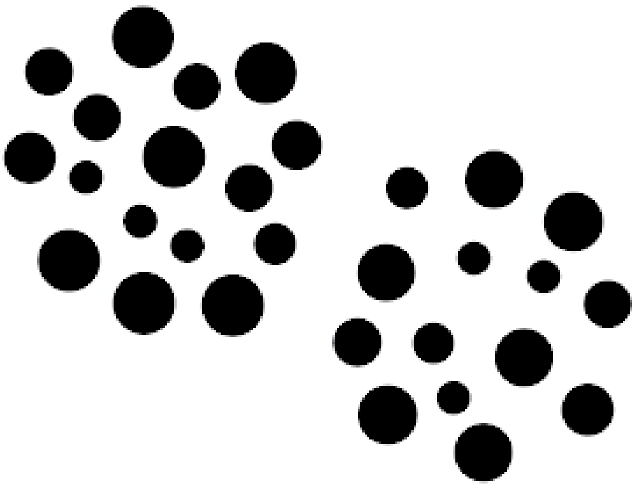

Figure 1

In today's world, we generate massive amounts of data every second—customer transactions, social media activity, medical records, and much more. However, data alone is not enough; we need methods to uncover meaningful patterns and insights. This is where **clustering** comes in.

Clustering is a fundamental technique in **unsupervised learning**, where the goal is to group similar data points together without prior labels or predefined categories. Unlike classification, where we already know the categories, clustering helps us discover hidden structures within data.

Clustering is widely used across multiple fields, providing valuable insights and automating processes. Some of its most common applications include:

- **Customer segmentation:** Businesses analyze customer behavior and group them into segments for personalized marketing, product recommendations, and customer retention strategies.
- **Image and pattern recognition:** Clustering techniques help organize large sets of images, recognize faces, and even classify handwritten characters.
- **Anomaly detection:** Financial institutions and cybersecurity experts use clustering to detect unusual patterns, such as fraudulent transactions or network intrusions.
- **Medical and biological research:** Scientists use clustering to group patients with similar symptoms, identify genetic similarities, and classify diseases.
- **Search engines and recommendation systems:** Clustering improves search results and suggests content by grouping similar webpages, articles, or products together.

At its core, clustering relies on measuring the similarity between data points. There are different approaches to achieve this, depending on the type of data and the desired outcome. Some of the most popular clustering algorithms include:

- **K-Means:** A simple and widely used algorithm that partitions data into a predefined number of clusters based on similarity.
- **Hierarchical Clustering:** Builds a tree-like structure of nested clusters, allowing for more flexible grouping.

- **DBSCAN:** Identifies clusters based on data density, making it useful for detecting irregularly shaped clusters and noise.
- **Gaussian Mixture Models (GMM):** A probabilistic approach that assumes data is generated from multiple overlapping distributions.

Clustering helps us transform complex, unstructured data into meaningful insights, making it a crucial tool for data scientists, researchers, and businesses alike. Whether we are identifying customer trends, detecting fraud, or advancing medical research, clustering allows us to find patterns in the chaos and make data-driven decisions.

K-Means

According to Raschka et al. (2022) [1], the k-means algorithm belongs to the category of **prototype-based clustering**, that means that each cluster is represented by a prototype, which is usually either the **centroid** (average) of similar points with continuous features, or the **medoid** (the most representative or the point that minimizes the distance to all other points that belong to a particular cluster) in the case of categorical features.

K-means is highly effective at detecting clusters that have a roughly spherical shape, making it a widely used clustering algorithm. However, one of its main limitations is that it requires the user to define the number of clusters, denoted as k, before running the algorithm. This necessity can be problematic because choosing an unsuitable value for k may lead to suboptimal results, where data points are grouped incorrectly or meaningful patterns in the dataset are not properly captured. Selecting the right number of clusters is crucial, as an overly high or low k value can distort the underlying structure of the data, ultimately reducing the accuracy and reliability of the clustering process.

We will work with the following synthetic data set for pedagogical purposes.

Our synthetic data

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generate synthetic data
X, y = make_blobs(n_samples=150,
                  n_features=2,
                  centers=3,
                  cluster_std=2,
                  shuffle=True,
                  random_state=42)

# Function to visualize the data
def plot_data(X):
    plt.figure(figsize=(7, 5))  # Adjust figure size

    plt.scatter(X[:, 0], X[:, 1],
                c='royalblue', marker='o',
                edgecolor='black', s=60, alpha=0.7)

    plt.xlabel('Feature 1', fontsize=12)
    plt.ylabel('Feature 2', fontsize=12)
    plt.title('Generated Data from make_blobs', fontsize=14, fontweight='bold')

    plt.grid(True, linestyle="--", alpha=0.6)
    plt.show()

# Call the visualization function
plot_data(X)
```

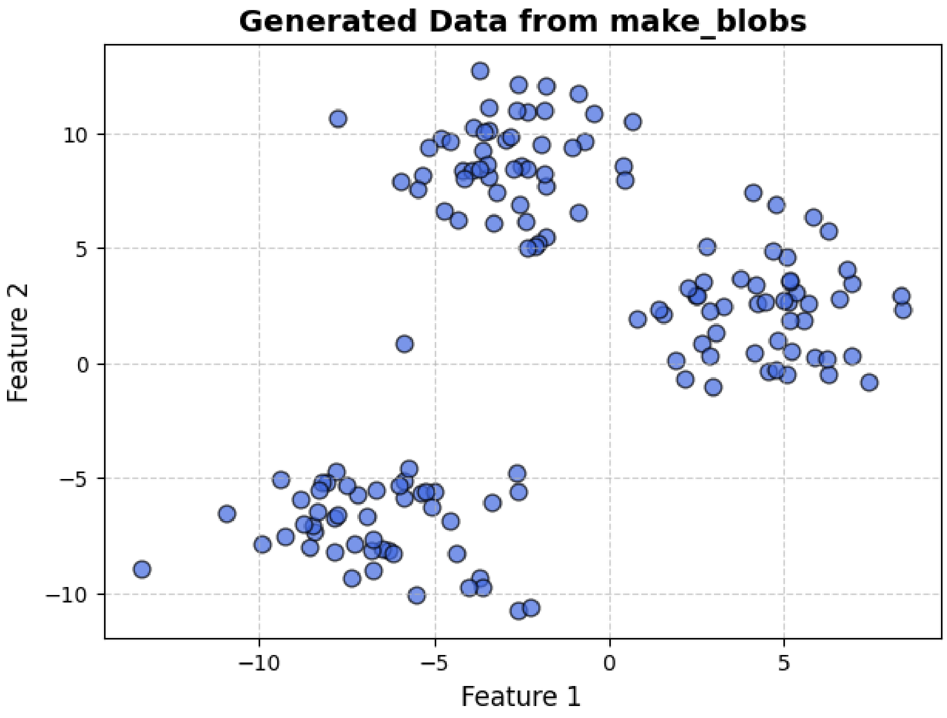

Figure 2: Synthetic Data

Our objective is to group the examples based on their feature similarities, which can be accomplished using the k-means algorithm. This process can be summarized in four key steps:

1. Randomly select k centroids from the examples, which will serve as the initial cluster centers.
2. Assign each example to the nearest centroid, μ_j, where j ranges from 1 to k, based on the proximity in the feature space.
3. Move the centroids to the center of the examples assigned to them, so that each centroid is better positioned in relation to its respective group.
4. Repeat steps 2 and 3 until the cluster assignments no longer change, or until a user-defined limit is reached, either in terms of maximum tolerance or the maximum number of iterations.

This iterative process allows the algorithm to refine the grouping of examples, reaching a clustering configuration that best reflects the similarities between the data's features.

Measuring similarity

Similarity can be thought of as the inverse of distance, and one of the most commonly used distance metrics for clustering examples with continuous features is the squared Euclidean distance between two points, $\mathbf{x}$ and $\mathbf{y}$, in an m-dimensional space.

$$d(\mathbf{x},\mathbf{y})^2 = \sum_{j=1}^{m}(x_j - y_j)^2 = ||\mathbf{x} - \mathbf{y}||_2^2$$

Based on this Euclidian distance metric, the k-means algorithm becomes a simple optimization problem, an iterative approach for minimizing the within-cluster **sum of squared errors (SSE)**, which is sometimes called **cluster inertia**:

$$SSE = \sum_{i=1}^{n}\sum_{j=1}^{k} w^{(i,j)}||\mathbf{x}^{(i)} - \mu^{(j)}||_2^2$$

$\mu^{(j)}$ is the representative point (centroid) for cluster j. $w^{(i,j)} = 1$ if the example, $x^{(i)}$, is in cluster j, or 0 otherwise.

sklearn KMeans

KMeans

```
class sklearn.cluster.KMeans(n_clusters=8, *, init='k-means++', n_init='auto',
↪  max_iter=300, tol=0.0001, verbose=0, random_state=None, copy_x=True,
↪  algorithm='lloyd')
```

Parameters:

- **n_clusters** `int`, default=8:
  The number of clusters to create, as well as the number of centroids to generate.
- **init** `{'k-means++', 'random'}`, callable or array-like of shape (n_clusters, n_features), default=`'k-means++'`:
  Method for initializing the centroids:
  - `'k-means++'`: Initializes cluster centroids by sampling based on an empirical probability distribution of each point's contribution to the overall inertia. This method accelerates convergence. The algorithm used is "greedy k-means++," which differs from traditional k-means++ by performing multiple trials at each step and selecting the best centroid from the trials.
  - `'random'`: Selects `n_clusters` random observations (rows) from the data to serve as initial centroids.
- **n_init** `'auto'` or int, default=`'auto'`:
  The number of times the k-means algorithm is executed with different centroid initializations.
- **max_iter** `int`, default=300:
  The maximum number of iterations for a single run of the k-means algorithm.
- **tol** `float`, default=1e-4:
  Relative tolerance with respect to the Frobenius norm of the difference in the cluster centers between two consecutive iterations, used to declare convergence.
- **verbose** `int`, default=0:
  Level of verbosity for the algorithm's output.
- **random_state** `int`, `RandomState` instance, or None, default=None:
  Controls random number generation for centroid initialization. Using an integer ensures deterministic results. See the Glossary for more details.
- **copy_x** `bool`, default=True:
  When precomputing distances, it is more numerically accurate to first center the data. If `copy_x` is True (default), the original data is not altered. If False, the original data is modified, then restored before returning the function's result, though small numerical differences may be introduced by subtracting and adding the data mean.
- **algorithm** `"lloyd", "elkan"`, default="lloyd":
  The k-means algorithm to use. The classical Expectation-Maximization (EM) style algorithm is "lloyd." The "elkan" variant can be more efficient for some datasets with well-defined clusters by using the triangle inequality. However, it is more memory-intensive due to the allocation of an extra array of shape (n_samples, n_clusters).

Let's apply it

```
from sklearn.cluster import KMeans

# Initialize KMeans with the specified parameters
km = KMeans(n_clusters=3,
            init='random',
            n_init=10,
            max_iter=300,
            tol=1e-04,
            random_state=0)
```

```
# Predict the clusters for the input data X
pred = km.fit_predict(X)

# Define a color palette for the clusters
colors = ['lightgreen', 'orange', 'lightblue']
markers = ['s', 'o', 'v']
labels = ['Cluster 1', 'Cluster 2', 'Cluster 3']

# Plot the points for each cluster
for i in range(3):
    plt.scatter(X[pred == i, 0], X[pred == i, 1],
                s=50, c=colors[i], marker=markers[i],
                edgecolor='black', label=labels[i])

# Plot the centroids
plt.scatter(km.cluster_centers_[:, 0], km.cluster_centers_[:, 1],
            s=250, marker='*', c='red', edgecolor='black', label='Centroids')

# Add labels and legend
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend(scatterpoints=1, loc='upper right', fontsize=12)

# Adjust the grid and layout
plt.grid(True, linestyle='--', alpha=0.6)
plt.tight_layout()
```

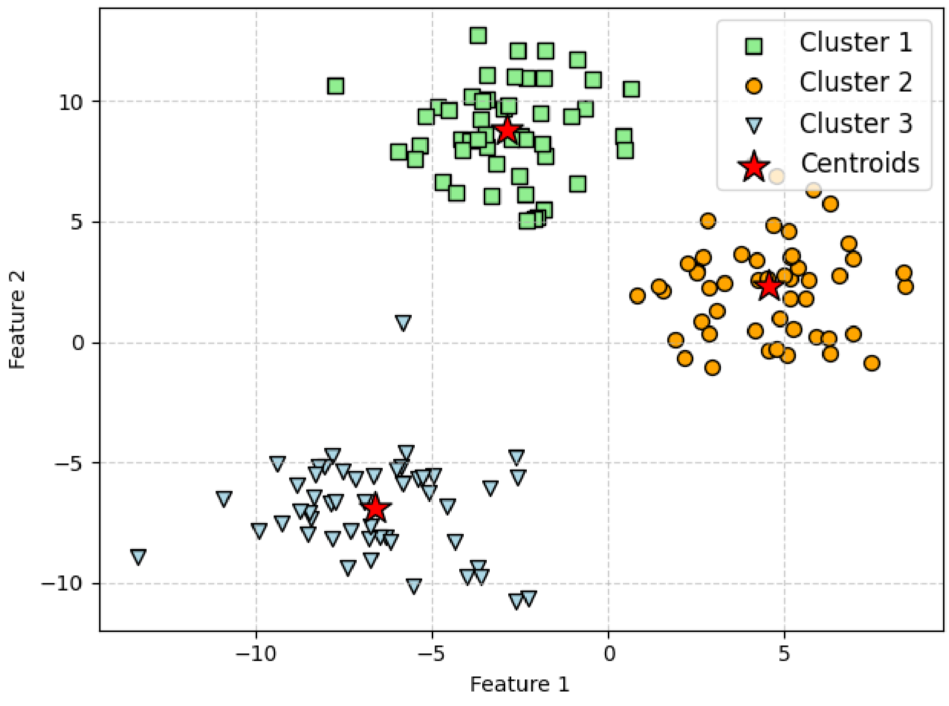

Figure 3

References

[1] Raschka, S., Liu, Y. & Mirjalili, V. (2022). *Machine Learning with PyTorch and Scikit-Learn*. Packt.