

# KMeans Under the Hood: From Euclidean Geometry to Clustering Algorithms

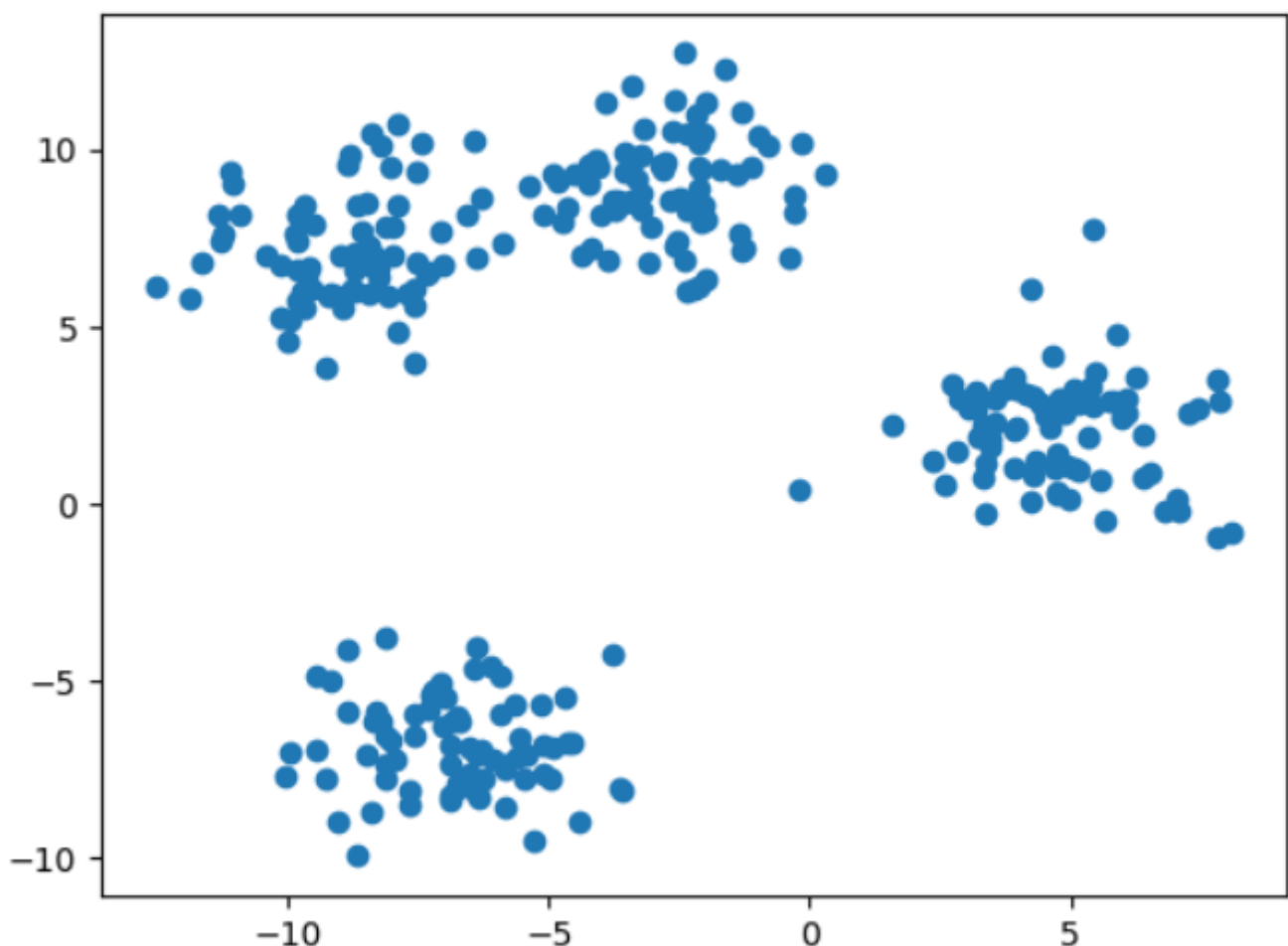
## 1. Introduction

Clustering is one of the most intuitive yet mathematically rich techniques in unsupervised learning. Among the algorithms that tackle this task, **KMeans** stands out for its simplicity, speed, and geometric clarity. But beneath the surface lies an elegant optimization problem rooted in **Euclidean distance minimization**—a process of iteratively reshaping our understanding of structure in unlabeled data.

In this post, we'll **pop the hood** and take a detailed look at how KMeans really works. From its **objective function grounded in Euclidean geometry** to the mechanics of **convergence and centroid updates**, we'll unpack the algorithm step by step. You'll see how to build it from scratch using NumPy, visualize how clusters form and evolve, and benchmark our implementation against scikit-learn's. Along the way, we'll also explore where KMeans performs well—and where its assumptions start to break down.

## 2. Understanding KMeans: From Intuition to Optimization

### 2.1 Intuition



Imagine that the scatterplot above represents the locations of your customers on a map. Your goal is to open **4 delivery hubs**, and you want to place them in a way that minimizes how far each customer needs to travel to their nearest hub.

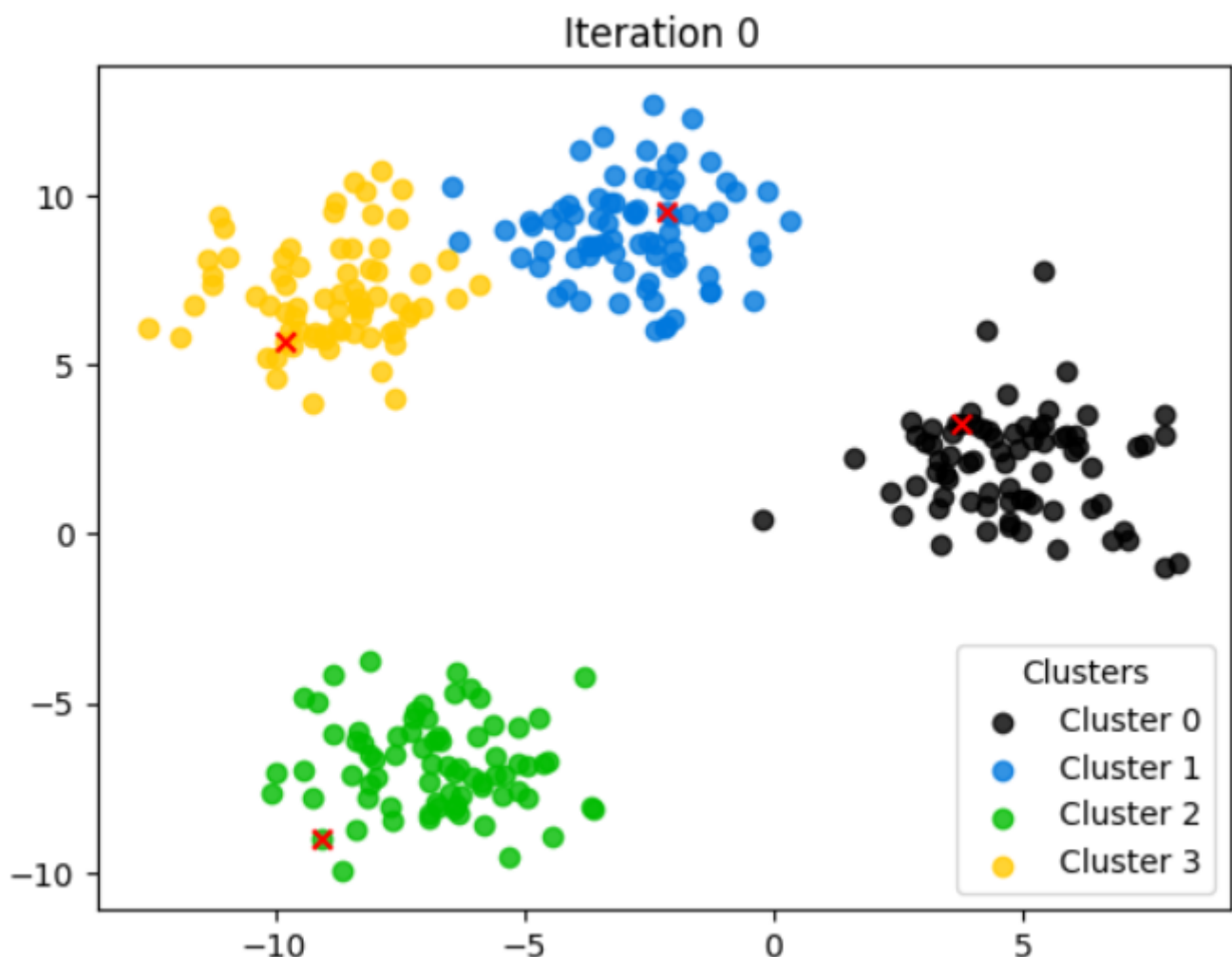
That's exactly what **KMeans** is designed to do. It finds natural groupings in the data by locating the **centroid** of each cluster—the center of each tightly packed group. These centroids act like hub locations that pull nearby data points toward them.

Here's how the process unfolds:

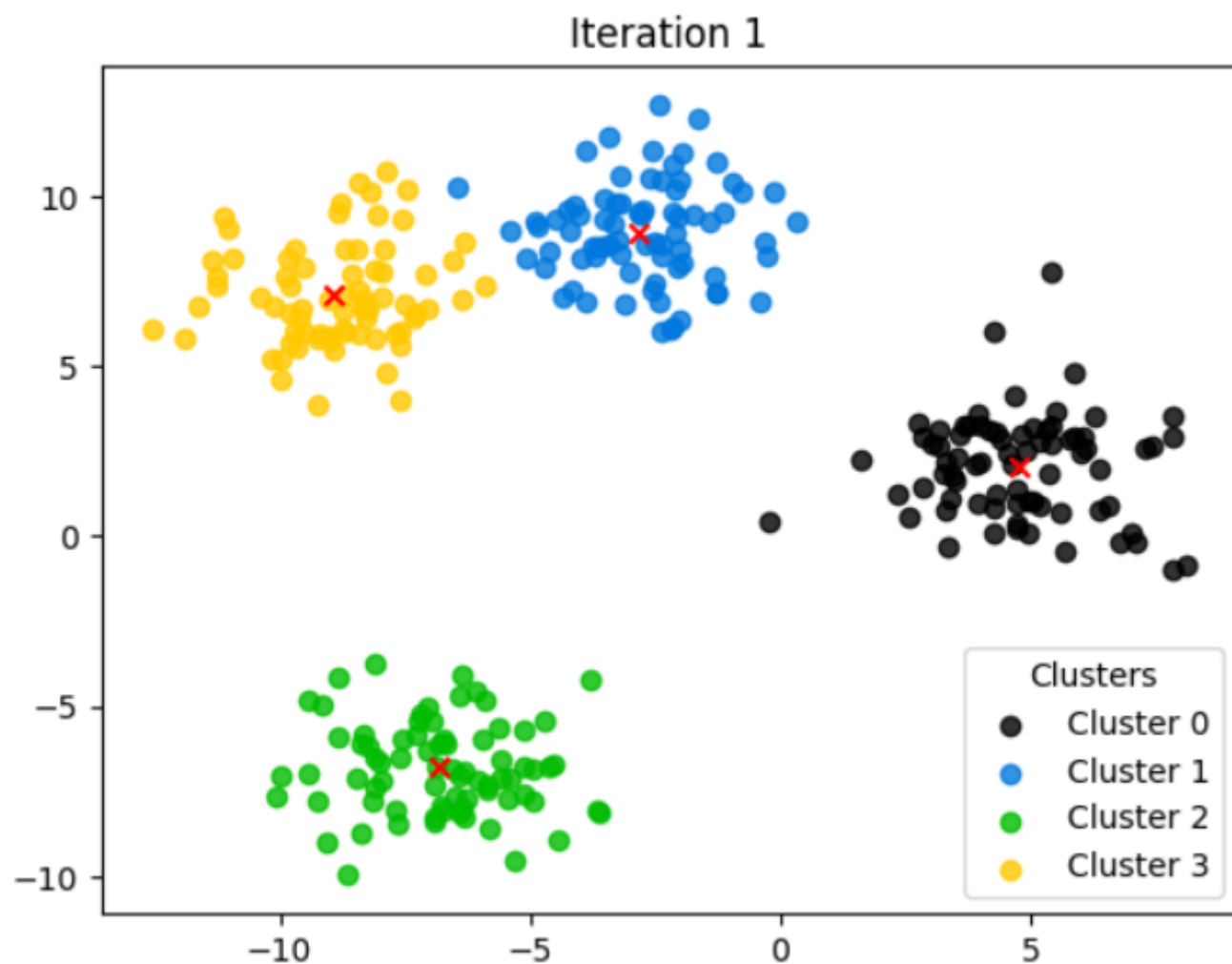
1. **Initialize** 4 hub locations (centroids) randomly.
2. **Assign** each customer to the nearest hub.
3. **Update** the location of each hub to the average position of its assigned customers.
4. **Repeat** this assign-update cycle until the hubs stop moving significantly.

Through this simple but powerful loop, KMeans uncovers the underlying structure in the data—forming clusters that are spatially compact and well-separated.

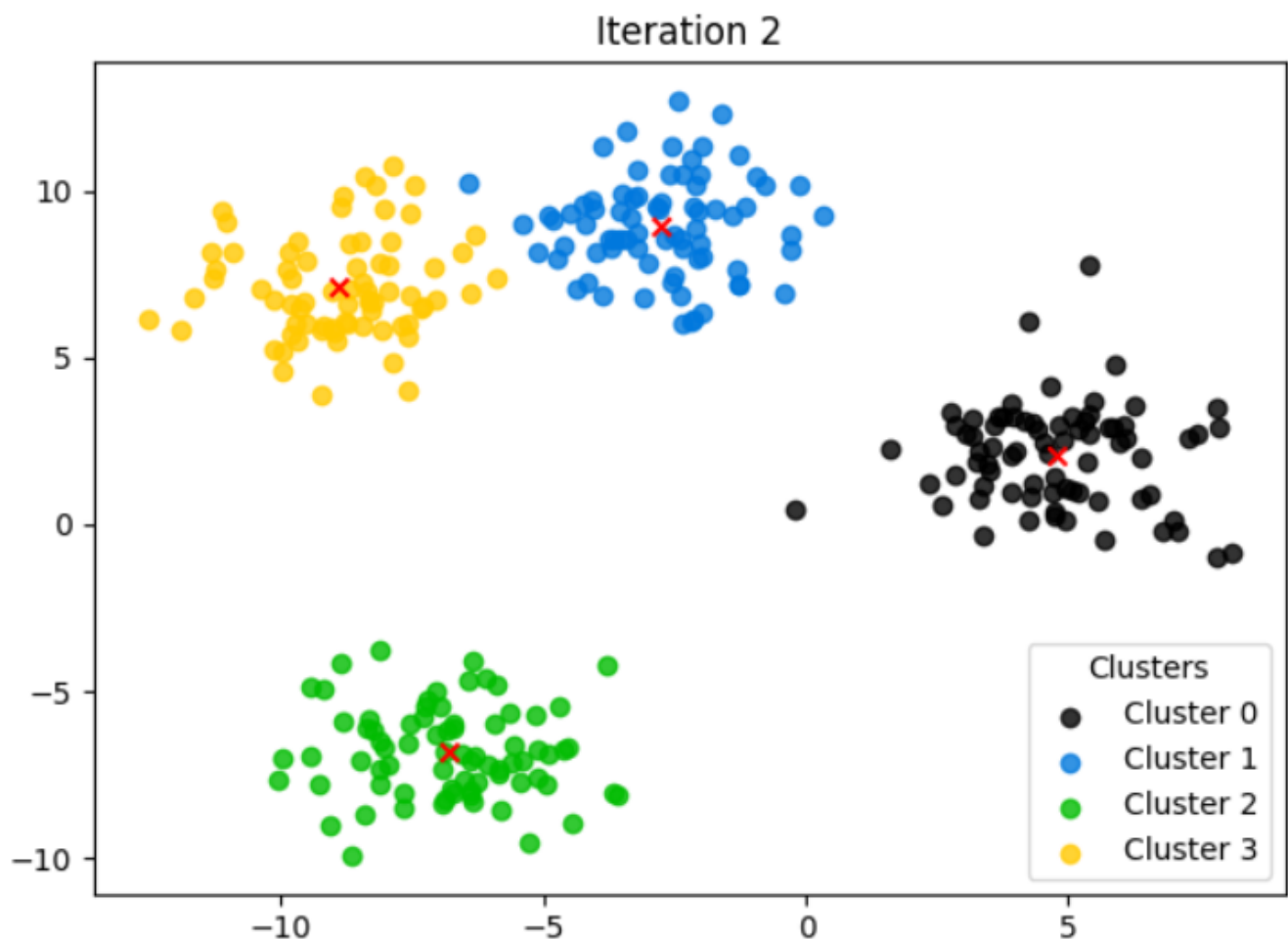
Here's a visual example of **KMeans in action**, performing the steps described above across **three iterations**. The red  $\times$  marks represent the centroids, while each color indicates a different cluster of customers:



In **iteration 0**, the centroids are placed randomly. KMeans assigns each customer to the nearest centroid, forming the initial clusters.



In **iteration 1**, the algorithm updates each centroid to be the **mean of its assigned customers**, then reassigns points based on these new positions.



By **iteration 2**, the centroids remain unchanged—no better locations were found, so the algorithm **converged**. At this point, KMeans considers the clustering complete, having reached a stable configuration that minimizes the total within-cluster distance.

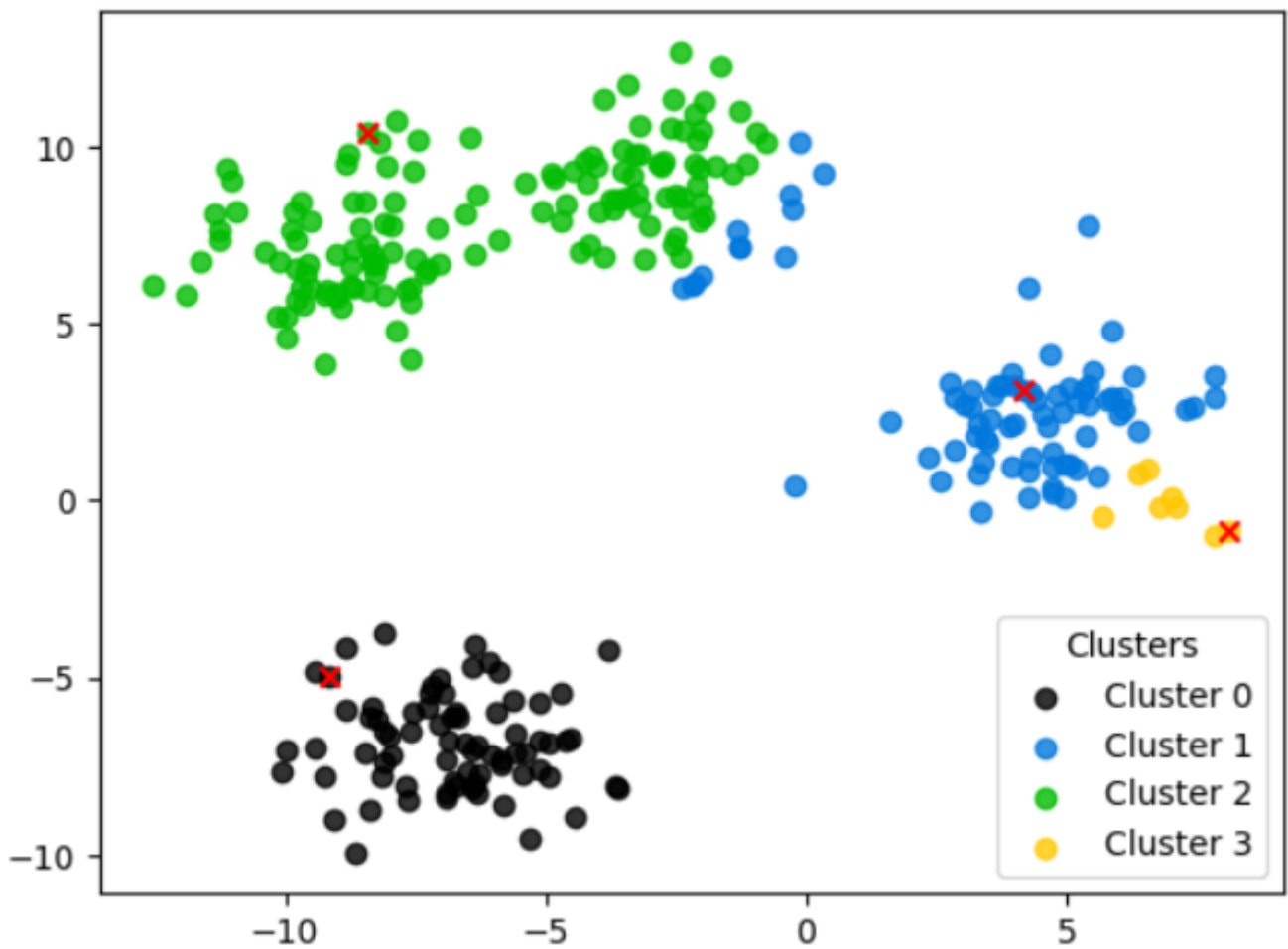
Well, I have a bit of a confession to make...

The algorithm you saw in the visual above isn't *exactly* the original KMeans. It's actually a slightly smarter variant called **KMeans++**.

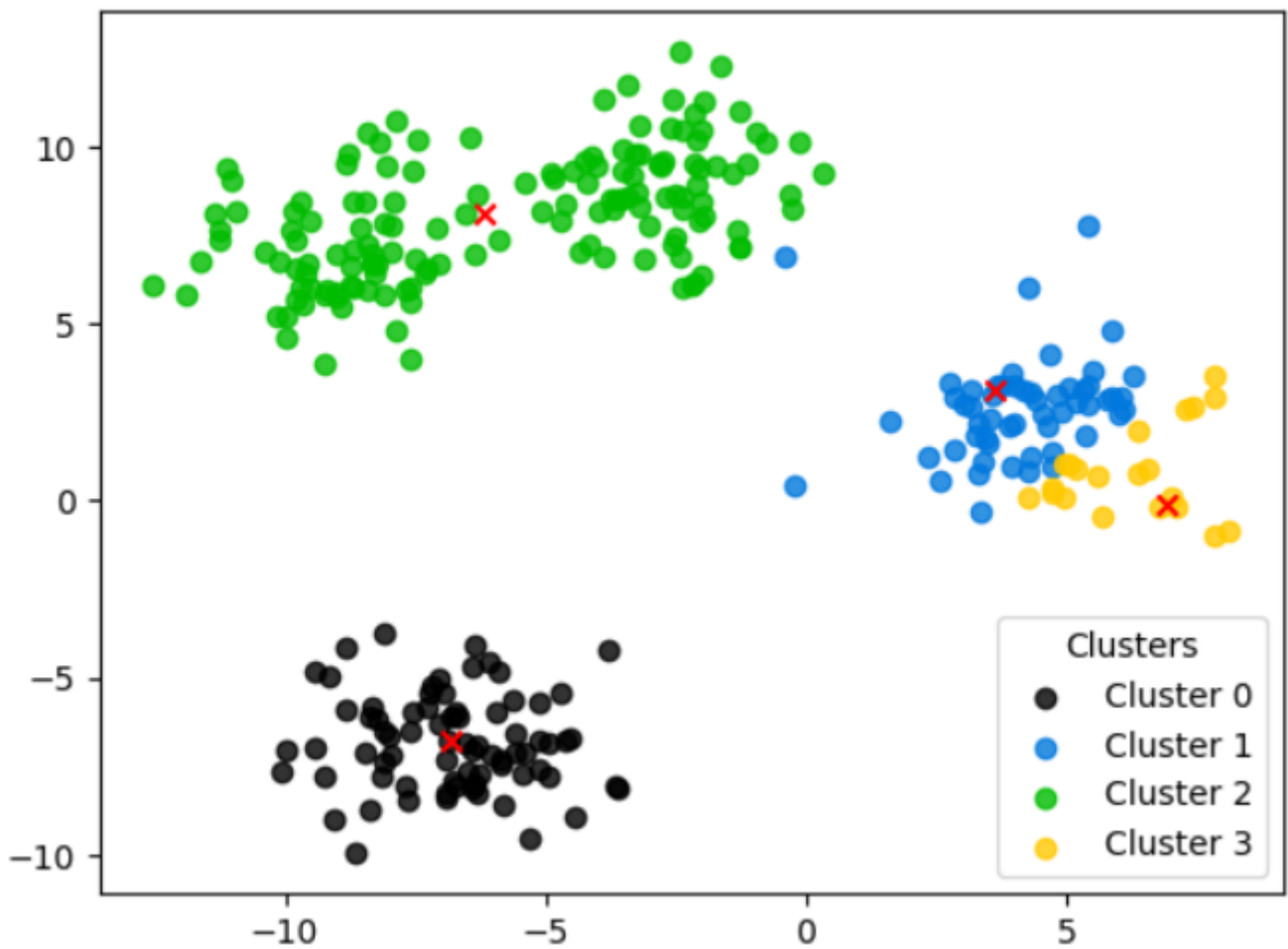
You see, vanilla KMeans is inherently **random**—it initializes centroids without any strategy, which makes it surprisingly inconsistent. The final clustering can vary significantly depending on where those initial centroids land. For example, if two centroids are randomly placed within the same true cluster, the algorithm can completely misrepresent the actual structure of the data.

Here's what that looks like in practice:

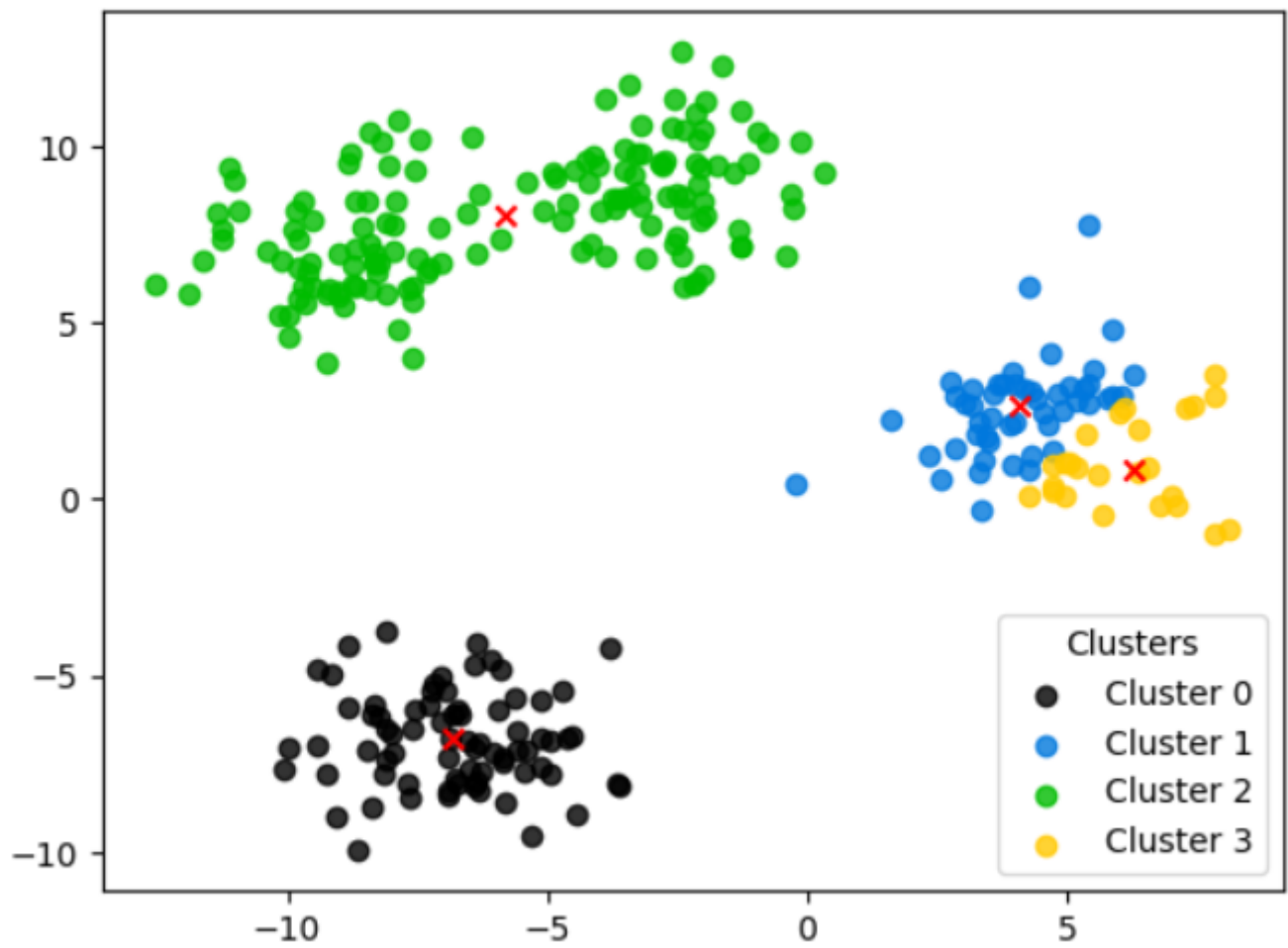
Iteration 0



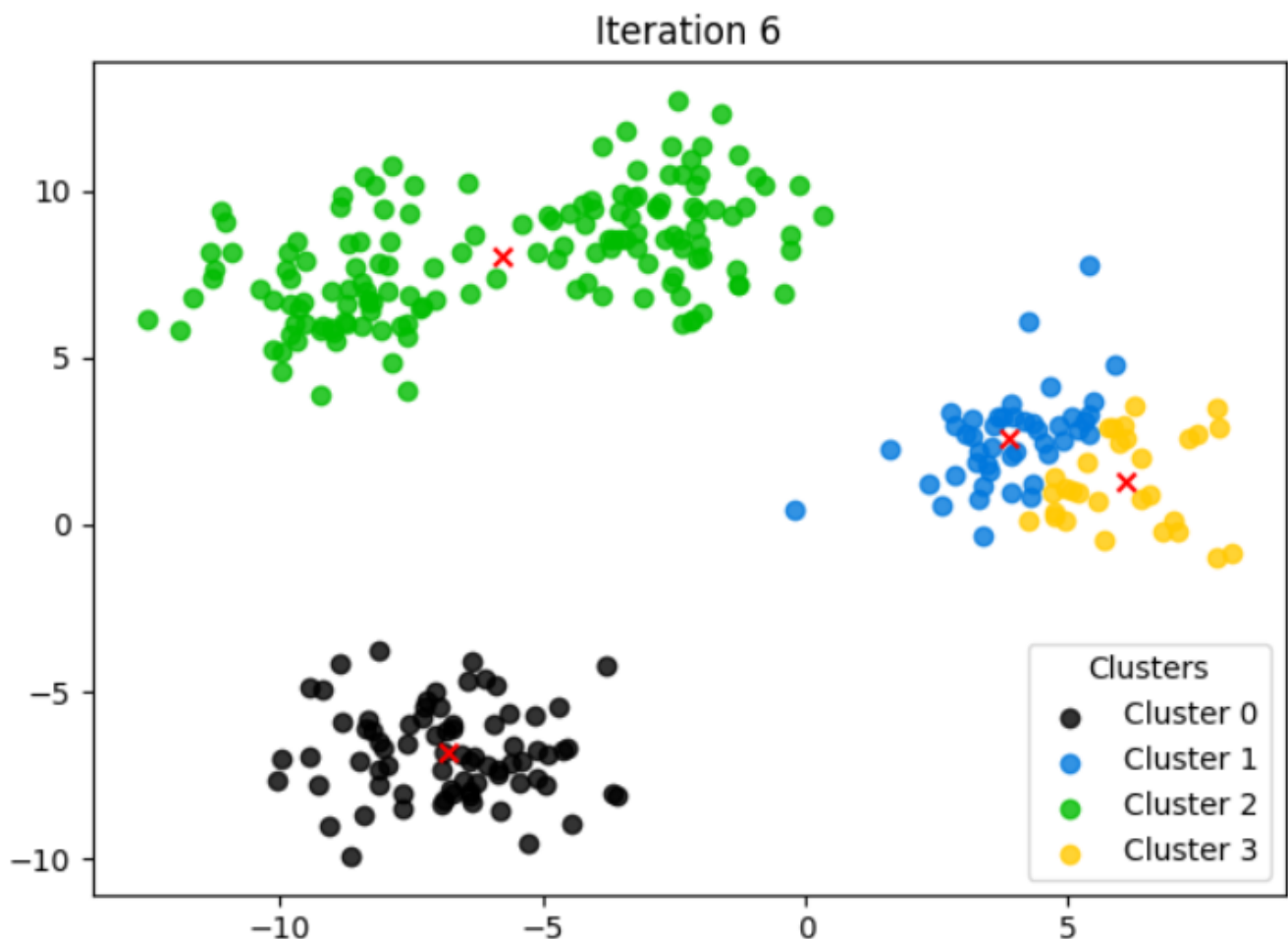
Iteration 1



Iteration 2



...



As you can see, due to the **poor initial placement of centroids in iteration 0**, the algorithm ends up in a **suboptimal configuration by iteration 6**—one that looks nothing like the correct clustering we saw earlier. The final result is drastically misaligned with the true structure of the data, all because of a bad start.

This randomness is exactly why **KMeans++** was introduced—to make smarter guesses when choosing the initial centroids. Let's take a closer look at how it works and why it makes a big difference.

KMeans++ improves the initialization step by **spreading out the initial centroids** in a way that's more informed than pure randomness. Instead of picking all centroids blindly, it carefully selects each new one based on how far it is from the ones already chosen.

Here's how it works:

1. **Pick the first centroid** randomly from the dataset.
2. For each remaining data point, **compute its distance to the nearest chosen centroid**.
3. Select the next centroid with **probability proportional to the square of that distance** (i.e., farther points are more likely to be picked).
4. Repeat steps 2–3 until you've chosen K centroids.

This strategy ensures that initial centroids are **well-separated**, reducing the chance of poor coverage or overlapping starting points.

And the result? **Faster convergence, lower distortion, and more consistent clustering outcomes**—especially when the data has well-defined clusters.

Now, **KMeans++ isn't perfect**—and it doesn't guarantee a good initialization every time. In fact, the *bad example* we looked at earlier was actually generated using **KMeans++**.

The key difference is that while it can still fail, it's **far more consistent** than vanilla KMeans. By spreading out the initial centroids intelligently, it **reduces the chances** of poor clustering outcomes, especially on datasets with clear structure. In practice, this usually translates to **faster convergence and better final clusters**.

So, while it doesn't eliminate randomness completely, it does a much better job at controlling it

## 2.2 The Math Behind KMeans

Once the intuition is clear, it's worth understanding **what KMeans is actually optimizing** — and why it works.

At its core, KMeans tries to minimize the total distance between each data point and the centroid of the cluster it's assigned to. This distance is measured using the **squared Euclidean norm**, and the total objective is called the **within-cluster sum of squares** (WCSS), a.k.a. **inertia**:

$$J = \sum_{k=1}^K \sum_{x_i \in C_k} ||x_i - \mu_k||^2$$

Where:

- $K$  is the number of clusters
- $C_k$  is the set of points assigned to cluster  $k$
- $\mu_k$  is the centroid (mean) of cluster  $k$

The algorithm tries to find a set of centroids and assignments that **minimize  $J$** .

KMeans optimizes this objective using an **iterative two-step process**:

### 1. Assignment step (Expectation-step):

Given current centroids  $\mu_k$ , assign each point  $x_i$  to the cluster with the nearest centroid:

$$c_i = \arg \min_k ||x_i - \mu_k||^2$$

### 2. Update step (Maximization-step):

Recompute each centroid as the mean of the points assigned to it:

$$\mu_k = \frac{1}{|C_k|} \sum_{x \in C_k} x$$



This is a form of **coordinate descent** — it keeps alternating between fixing the assignments and updating the centroids, and in each step, the objective function  $J$  **monotonically decreases or stays the same**.

### Why It Converges (But Not Necessarily Well):

Because the number of possible cluster assignments is **finite**, and each iteration reduces the total error  $J$ , KMeans is guaranteed to **converge in a finite number of steps**.

However, that doesn't mean it finds the global minimum. KMeans often converges to **local minima**, which is why **initialization** matters so much — and why **KMeans++** helps.

Note that the KMeans objective is **non-differentiable** due to the discrete assignment step—each point either belongs to cluster **A** or **B**, with no continuous gradient in between. This hard assignment makes it impossible to apply standard gradient descent. Instead, KMeans uses an **alternating optimization strategy**, known as **coordinate descent**, which handles this discrete-continuous structure by optimizing one component at a time: first the assignments, then the centroids.

## 3. From Theory to Code: KMeans

Let's put theory into practice and build our own `KMeans` class from scratch — complete with **KMeans++ initialization**, **inertia tracking**, and **iteration history** for visualizations.

```
import numpy as np

from .base_model import BaseModel

from utils.helpers import validate_transform_input


class KMeans(BaseModel):

    def __init__(self, n_clusters=8, max_iter=int(1e9), tol=0):

        self._n_clusters = n_clusters

        self._max_iter = max_iter

        self._tol = tol

        self._inertia = None
```

```

self._centroids = None

self._labels = None

self._centroids_over_iters = []

self._labels_over_iters = []

def __compute_squared_euclidean_dist(self, X, centroid):

    return np.sum((X - centroid)**2, axis=1)

def __compute_inertia(self, X, centroids, labels):

    inertia = 0

    for i in range(self._n_clusters ):

        inertia += np.sum(self.__compute_squared_euclidean_dist(X[labels
== i], centroids[i]))

    return inertia

def __compute_initial_centroids(self, X):

    m = X.shape[0]

    centroids = []

    D = np.full(m, np.inf)

    init_centroid_idx = np.random.choice(m)

    init_centroid = X[init_centroid_idx]

    centroids.append(init_centroid)

    for _ in range(self._n_clusters - 1):

        D = np.minimum(D, self.__compute_squared_euclidean_dist(X,
centroids[-1]))

```

```

        probs = D / np.sum(D)

        new_centroid_idx = np.random.choice(m, p=probs)

        new_centroid = X[new_centroid_idx]

        centroids.append(new_centroid)

    return np.array(centroids)

def __compute_labels(self, X, centroids):

    distances = np.vstack([self.__compute_squared_euclidean_dist(X,
centroids[i])

                            for i in range(self._n_clusters)])

    return np.argmin(distances, axis=0)

def __compute_centroids(self, X):

    for _ in range(int(self._max_iter)):

        centroids = []

        for i in range(self._n_clusters):

            cluster_points = X[self._labels == i]

            if len(cluster_points) == 0:

                centroids.append(X[np.random.choice(X.shape[0])])

            else:

```

```

        centroids.append(np.mean(cluster_points, axis=0))

    new_labels = self.__compute_labels(X, centroids)

    new_inertia = self.__compute_inertia(X, centroids, new_labels)

    if self._inertia > new_inertia and abs(self._inertia -
new_inertia) > self._tol:

        self._inertia = new_inertia

        self._centroids = np.array(centroids)

        self._labels = new_labels

        self._centroids_over_iters.append(centroids)

        self._labels_over_iters.append(new_labels)

    else:

        break

def fit(self, X):

    X, _ = self._validate_transform_input(X)

    self._centroids = self.__compute_initial_centroids(X)

    self._centroids_over_iters.append(self._centroids)

    self._labels = self.__compute_labels(X, self._centroids)

    self._labels_over_iters.append(self._labels)

    self._inertia = self.__compute_inertia(X, self._centroids,
self._labels )

    self.__compute_centroids(X)

```

```

        self._centroids_over_iters = np.array(self._centroids_over_iters)

        self._labels_over_iters = np.array(self._labels_over_iters)

    return self

def transform(self, X):

    X, _ = self._validate_transform_input(X)

    distances = np.zeros((X.shape[0], self._n_clusters))

    for i in range(self._n_clusters):

        distances[:,i] = self.__compute_squared_euclidean_dist(X,
self._centroids[i])

    return distances

def fit_transform(self, X):

    self.fit(X)

    return self.transform(X)

def predict(self, X):

    X, _ = self._validate_transform_input(X)

    return self.__compute_labels(X, self._centroids)

def fit_predict(self, X):

    self.fit(X)

    return self.predict(X)

```

### 3.1 Initializing the Model

The constructor allows us to configure:

- `n_clusters` : Number of clusters to find (i.e., K).
- `max_iter` : Maximum number of iterations allowed before stopping.
- `tol` : Minimum improvement in inertia required between iterations to continue.
- Internal variables like `_centroids`, `_inertia`, `_labels`, `_centroids_over_iters`, and `_labels_over_iters` are used to track:
  - Final cluster centers
  - Final and intermediate assignments
  - Inertia at each step (for convergence and evaluation)
  - Iteration history for plotting clustering evolution

### 3.2 Measuring Distance and Inertia

Two internal helper methods handle key mathematical operations:

- `__compute_squared_euclidean_dist` calculates the squared L2 distance between a set of points and a given centroid.
- `__compute_inertia` calculates the **total within-cluster sum of squared distances**, summed **across all clusters**. For each cluster, it adds up the squared distances between the points assigned to that cluster and their centroid. This total is the **KMeans cost function**, also known as **inertia** or **WCSS (within-cluster sum of squares)**.

### 3.3 KMeans++ Initialization

The method `__compute_initial_centroids` implements the **KMeans++** strategy to pick better starting centroids:

- The first centroid is chosen randomly.
- For each new centroid, we compute the distance of all points to the **nearest** existing centroid.
- The next centroid is then sampled **with probability proportional to the squared distance**, encouraging spread and reducing bad starts.
- This method improves convergence and consistency significantly compared to naive random initialization.

### 3.4 Assigning Clusters

The method `__compute_labels` assigns each data point to its closest centroid:

- It computes the distance from every point to every centroid.
- Each point is then assigned the label of the centroid with the smallest distance.
- The result is a 1D array of labels (same length as the dataset) that maps each point to its cluster.

### 3.5 Updating Centroids and Checking for Convergence

The method `__compute_centroids` handles the **core iterative loop** of the algorithm:

- For each cluster, compute the new centroid as the **mean** of all points assigned to it.
- If a cluster has no points assigned (which can happen), it randomly reinitializes that centroid to avoid crashes.
- After new centroids are computed, it:
  - Recomputes labels
  - Calculates the new inertia
  - Compares it with the previous inertia
  - If the improvement is less than the tolerance `tol`, it terminates early.
- It also logs centroid and label snapshots across iterations for optional visualization later.

### 3.6 Training the Model

The `fit` method puts it all together:

- Validates the input data.
- Uses **KMeans++** to compute the initial centroids.
- Assigns initial labels based on those centroids.
- Calculates the initial inertia.
- Enters the iterative update loop to refine the centroids and assignments.
- After training completes, it stores the full iteration history as NumPy arrays for easier access and visualization.

### 3.7 Making Predictions

The `predict` method allows you to assign cluster labels to **new, unseen data**:

- It computes the distance from each point to all centroids.
- Returns the index of the closest centroid for each point.

### 3.8 Distance Matrix and Utility Methods

Additional utility methods:



- `transform` returns the **distance matrix** between each point and all centroids. This can be used in pipelines or for scoring.
- `fit_predict` is a shorthand that combines `fit()` and `predict()` — useful for quick clustering runs without needing to call both explicitly.

## 4. Custom vs scikit-learn: A Side-by-Side Comparison

To validate my custom `KMeans` implementation, I compared it against `sklearn.cluster.KMeans`, the industry-standard implementation used in most practical applications.

Since both models use the same core algorithm — including **inertia minimization**, **Euclidean distance**, and **iterative updates** — they are directly comparable in terms of clustering quality, convergence behavior, and even centroid placement.

I used two types of datasets for comparison:

-  The `make_blobs` dataset for clear, well-separated clusters visualization and evaluation.
-  A **real-world dataset** (`digits`) to test label alignment and clustering evaluation metrics.

### 4.1 Predictions & Evaluation

I compared the models based on:

- Inertia, silhouette, and adjusted-rand scores
- Silhouette score chart
- Visual clustering similarity

#### `make_blobs` Data Evaluation

##### Inertia, silhouette, and adjusted-rand scores

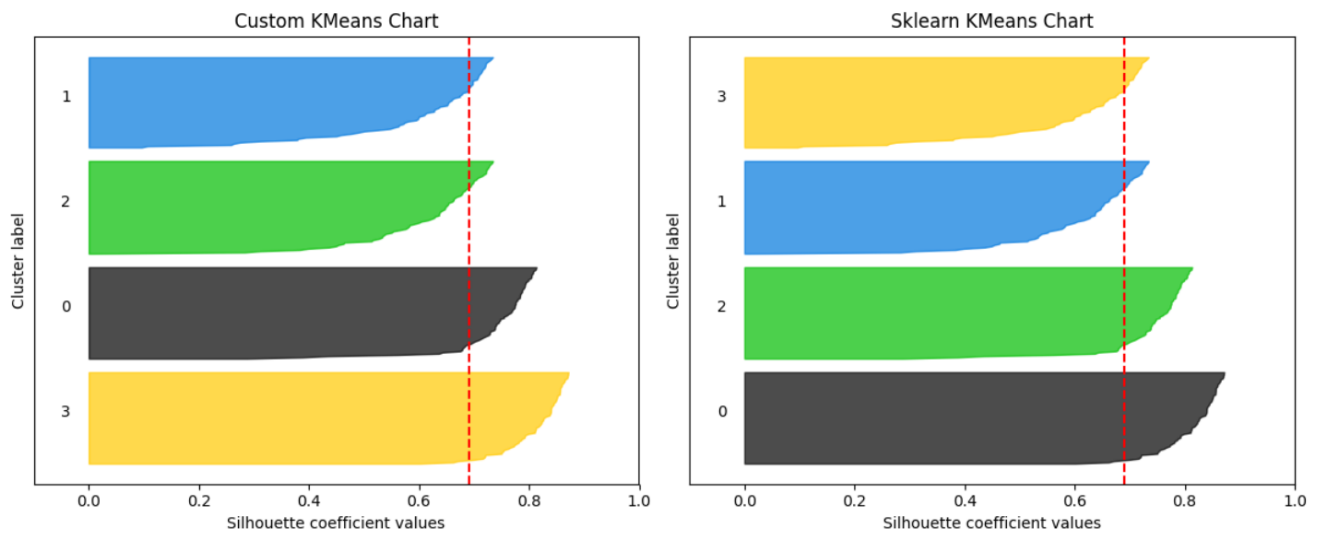
```
Custom KMeans Inertia: 1251.352629870747
Sklearn KMeans Inertia: 1251.3526298707468
```

	Model	Silhouette	Adjusted-Rand
0	Custom KMeans Evaluation	0.691248	0.991081

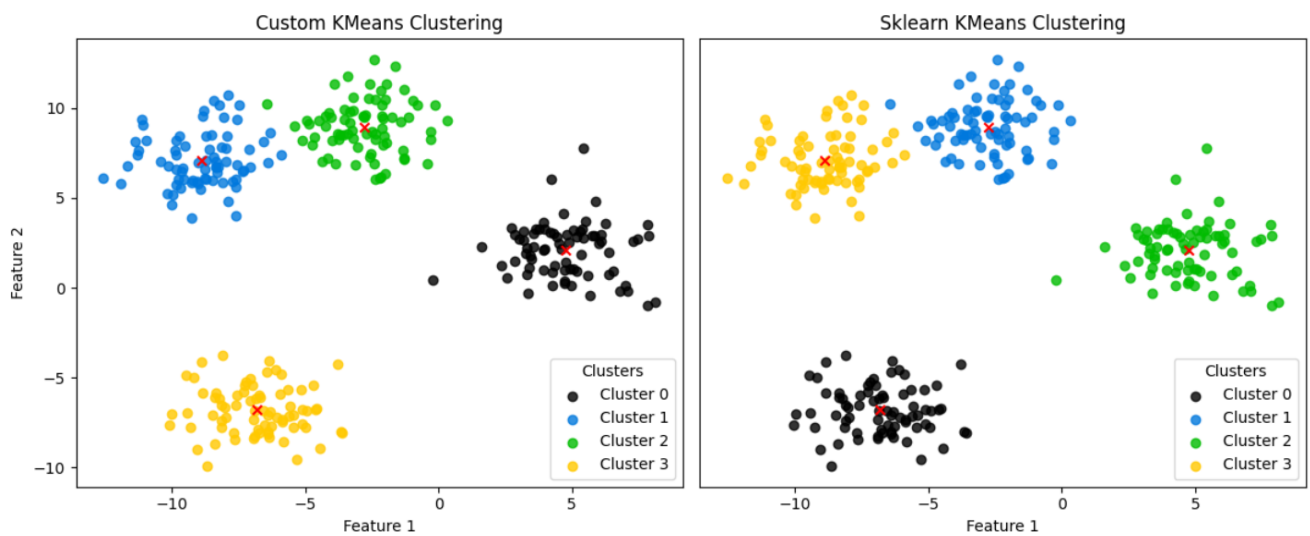
	Model	Silhouette	Adjusted-Rand
0	Sklearn KMeans Evaluation	0.691248	0.991081

##### Silhouette score chart





## Visual clustering similarity



## load\_digits Data Evaluation

### Trial 1

```
Adjusted-Rand Score Between Both Predictions: 0.5574111014360192
Adjusted-Rand Score Between True Labels and Custom Predictions: 0.5719577870038048
Adjusted-Rand Score Between True Labels and Sklearn Predictions: 0.5718944198902352
Silhouette Score for Custom Predictions: 0.18582119555398288
Silhouette Score for Sklearn Predictions: 0.13933006181248417
```

### Trial 2

```
Adjusted-Rand Score Between Both Predictions: 0.7634737113790014
```

```
Adjusted-Rand Score Between True Labels and Custom Predictions: 0.5770588369943552
```

```
Adjusted-Rand Score Between True Labels and Sklearn Predictions: 0.614156610452878
```

```
Silhouette Score for Custom Predictions: 0.15829919686829824
```

```
Silhouette Score for Sklearn Predictions: 0.18537466454440965
```

### Trial 3

```
Adjusted-Rand Score Between Both Predictions: 0.7642698262461438
```

```
Adjusted-Rand Score Between True Labels and Custom Predictions: 0.6619393225238155
```

```
Adjusted-Rand Score Between True Labels and Sklearn Predictions: 0.6562333024998562
```

```
Silhouette Score for Custom Predictions: 0.1821391476231858
```

```
Silhouette Score for Sklearn Predictions: 0.18813355171533816
```

As you can see across all `make_blobs` evaluations, both models performed **identically** in terms of inertia, silhouette scores, adjusted Rand index, and even visual cluster separation — confirming that the core logic and convergence behavior of my implementation is sound.

However, when tested on the more complex `load_digits` dataset, we start to see some **variance in performance** across different trials. In one run, my model slightly outperformed scikit-learn's; in another, scikit-learn pulled ahead; and in the third, they performed nearly identically.

This subtle back-and-forth is a great reminder of an important fact:

**KMeans is inherently non-deterministic**, and even with smarter initialization like `KMeans++`, different centroid seeds can lead to slightly different local optima — especially on high-dimensional or noisy datasets.

In real-world use cases, this variability is often managed by running KMeans multiple times with different initializations and selecting the result with the lowest final inertia (`n_init` in scikit-learn). My implementation is designed to support that behavior in future extensions, but even in its current form, it holds its own **remarkably well** — both in terms of clustering quality and overall reliability.

## 4.2 Bonus Test: Are KMeans Clusters Semantically Meaningful?

Up to this point, we've compared clustering quality using metrics like inertia, silhouette score, and visual alignment. But there's a deeper question worth asking:

**Do the clusters actually represent real, meaningful structure in the data?**

To explore this, I ran a simple but telling experiment using the `load_digits` dataset:

1. I fit my custom `KMeans` model with `n_clusters=30` on the dataset.
2. For each cluster, I selected the single **data point closest to the cluster centroid** — in other words, the most "representative" digit for that group.
3. I **manually labeled** these 30 representatives by visually inspecting the images.
4. I trained a `LogisticRegression` classifier on just these 30 labeled points.
5. Finally, I used the trained classifier to predict labels for the **entire digits dataset**, and evaluated the accuracy.

Here are the 30 representative digits I inspected and manually labeled:



No, they might not be in the 1080p full HD quality we would like to see — but they were sharp enough to guide a classifier through thousands of digits.

Despite using only 30 hand-labeled examples (one per cluster), the classifier achieved an accuracy of **89%** on the full dataset.

This is a powerful result: it shows that the clusters discovered by KMeans are not only geometrically compact, but also **semantically meaningful**. The algorithm was able to

uncover structure in the data that **closely mirrors the actual digit classes** — all without seeing a single label during clustering.

This kind of test turns KMeans from an abstract optimization routine into a practical, interpretable learning tool. It's a great sanity check — and a strong endorsement of the quality of the clustering itself.

## 5. Limitations of KMeans: When Geometry Gets in the Way

While KMeans is elegant and effective, it comes with a set of important assumptions — and they don't always hold in real-world data.

Here are some of its core limitations:

### 5.1 Assumes Spherical Clusters

KMeans assumes that all clusters are **roughly spherical** and equally sized. It works by minimizing Euclidean distance to the nearest centroid, which inherently draws **circular (or hyperspherical)** boundaries.

This becomes a problem when:

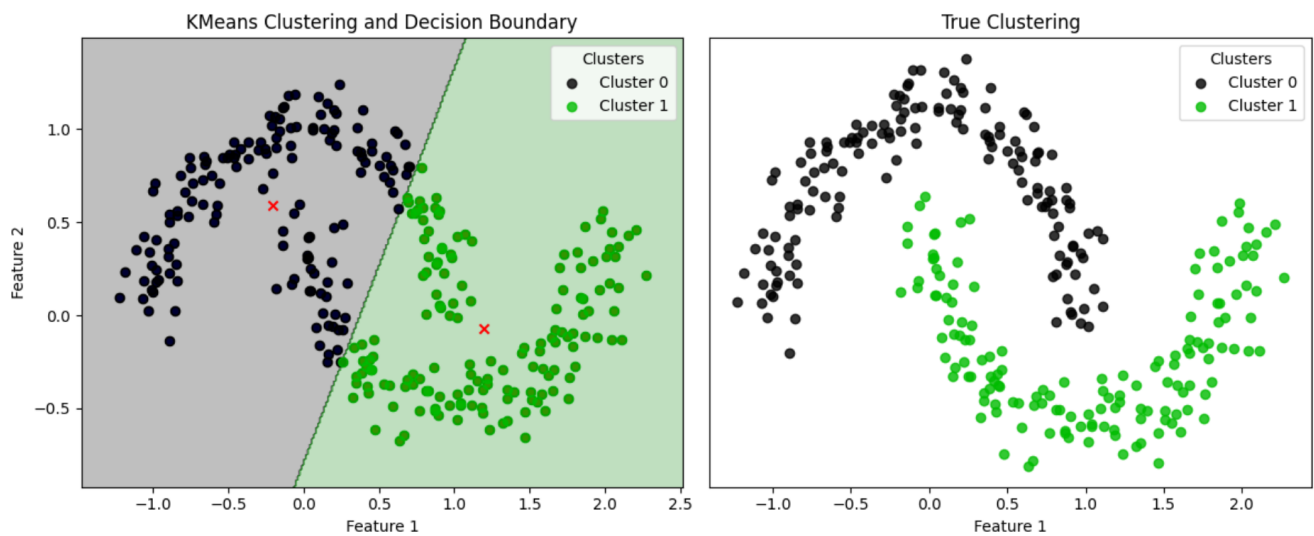
- Clusters are **elongated** or have different densities
- There is **variance in cluster size**

### 5.2 Breaks on Non-Convex Shapes

KMeans struggles with non-convex clusters like the classic `make_moons` or `make_circles` datasets:

- It will often slice through curved shapes in awkward ways.
- Since it relies on distance to a fixed center, it cannot follow curved boundaries or capture disconnected regions.

Here's a visual example on the `make_moons` dataset:



We can observe that KMeans forced the data into **straight-line partitions**, ignoring the true structure entirely.

### 5.3 Hard Assignments Only

KMeans gives each point a **hard assignment** — you're either in one cluster or another. There's no concept of **uncertainty** or **probability**.

This can be limiting in cases where:

- Data points lie between multiple clusters
- You want to model soft boundaries or overlaps
- You're dealing with noisy, ambiguous points

### 5.4 Sensitive to Initialization

Even with KMeans++, the algorithm is still vulnerable to **local minima** and **sensitive to the number of clusters  $K$** . You often have to:

- Run it multiple times ( `n_init > 1` )
- Experiment with different  $K$  values (e.g., using silhouette analysis or the elbow plots)

### 5.5 Bottom Line

KMeans is powerful, fast, and easy to implement — but it's not the right tool for every dataset. When you need more flexibility, better shape modeling, or soft probabilistic assignments, it's time to reach for a more expressive model.

## 6. So, Where Do We Go From Here?

I had a lot of fun building KMeans from scratch. It's such a classic — clean math, beautiful geometry, and surprisingly effective clustering, even with zero supervision.

But it's also... a bit rigid.

The moment you give it curved data or overlapping shapes, it panics and draws weird straight lines through the middle. It's trying its best, but it just wasn't made for nuance.

So in the next post, we're giving clustering a brain.

We'll talk about **Gaussian Mixture Models** — a method that doesn't assign a point to one cluster, but tells you how likely it is to belong to all of them.

Probabilities, soft assignments, and elliptical shapes. Clustering just got real.

## Connect & Explore More

If you found this implementation interesting or want to explore the full code, feel free to check out the repository:

 **GitHub:** <https://github.com/EyadMostafa/ml-from-scratch>

 **LinkedIn:** [www.linkedin.com/in/eyad-mostafa-813b11206](https://www.linkedin.com/in/eyad-mostafa-813b11206)

I'm always happy to connect, chat machine learning, or get feedback!