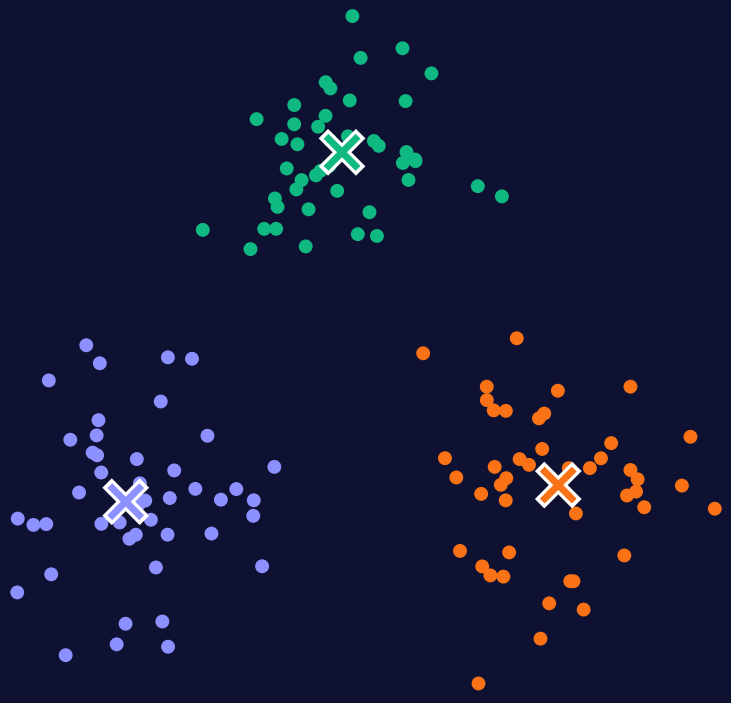
K-Means Clustering

Notes, Python code and interview questions



Code and Debug

codeanddebug.in/blog | YouTube: @codeanddebug

1. The idea

K-means clustering is an unsupervised algorithm that splits data into K groups (clusters) so that points in the same cluster are close together. Each cluster has a centre, the centroid, which is the mean of its points. Every point belongs to the cluster of its nearest centroid. You choose K; the algorithm finds the groups. No labels are needed.

2. Key terms

Term	Meaning
Cluster	A group of points that are close to each other.
K	The number of clusters, chosen before running the algorithm.
Centroid	The centre of a cluster: the mean of its points.
Inertia (WCSS)	Sum of squared distances from each point to its own centroid. Lower = tighter clusters.
Convergence	Assignments stop changing, so the algorithm stops.

3. The algorithm (Lloyd's algorithm)

- **Choose K**.
- **Initialise** K centroids (random points, or better, k-means++).
- **Assign** every point to its nearest centroid (Euclidean distance).
- **Update** each centroid to the mean of its assigned points.
- **Repeat** assign and update until no point changes cluster.

Objective (minimised by both steps):

```
WCSS = Σ_j Σ_(x in cluster j) ||x − c_j||²
```

K-means always converges because WCSS never increases, but only to a **local minimum** that depends on the starting centroids.

4. Solved numerical example (K = 2)

Point	A	B	C	D	E	F	G
(x, y)	(1,1)	(1.5,2)	(3,4)	(5,7)	(3.5,5)	(4.5,5)	(3.5,4.5)

- Start: c1 = A (1, 1), c2 = D (5, 7).
- Iteration 1: C is 3.61 from both (tie goes to cluster 1). Clusters {A, B, C} and {D, E, F, G}. New centroids (1.83, 2.33) and (4.12, 5.38).
- Iteration 2: C is now 2.03 from c1 and 1.78 from c2, so it moves to cluster 2. Clusters {A, B} and {C, D, E, F, G}. New centroids (1.25, 1.5) and (3.9, 5.1).
- Iteration 3: no point changes cluster. Converged.

5. Choosing K

- **Elbow method:** plot WCSS against K and pick the bend, after which extra clusters reduce WCSS only a little. WCSS always falls as K grows.
- **Silhouette score:** s = (b − a) / max(a, b), where a = mean distance to own cluster, b = mean distance to nearest other cluster. Range −1 to 1; pick the K with the highest average.
- Check that the clusters are useful for the problem, not just the scores.

6. k-means++ initialisation

- Pick the first centroid at random from the data.
- Pick each next centroid with probability proportional to D^2, its squared distance from the nearest centroid so far.
- Spreads centroids out, so poor local minima are much rarer. Default in scikit-learn; combine with n_init restarts.

7. KMeans cheat sheet (scikit-learn)

Parameter / attribute	Meaning
`n_clusters`	K (default 8)
`init`	`"k-means++"` (default) or `"random"`
`n_init`	Number of restarts; best run kept
`max_iter`	Max rounds per run (default 300)
`random_state`	Repeatable results
`cluster_centers_`	Centroids
`labels_`	Cluster of each training point
`inertia_`	WCSS
`predict(X)`	Nearest centroid for new points

8. Limitations and alternatives

Problem	What to do
Must choose K	Elbow, silhouette, domain knowledge
Different feature scales	StandardScaler before k-means
Bad starting centroids	k-means++ and n_init > 1
Outliers pull the mean	Remove them, or use K-medoids
Curved or uneven clusters	DBSCAN, Gaussian mixtures, spectral clustering
Categorical data	k-modes or k-prototypes
Very large data	MiniBatchKMeans

9. K-means vs KNN

	K-means	KNN
Type	Unsupervised clustering	Supervised classification/regression
K means	Number of clusters	Number of neighbours
Labels needed	No	Yes

10. Code examples (all tested)

Customer segmentation with KMeans

PYTHON

```
import numpy as np
from sklearn.cluster import KMeans

# 200 mall customers: monthly income (thousand rupees) and spending score (1 to 100)
rng = np.random.default_rng(42)
groups = [(25, 20, 30), (25, 80, 40), (55, 50, 50), (85, 20, 40), (85, 80, 40)]  # income, score, count
X = np.vstack([np.column_stack([rng.normal(inc, 7, n), rng.normal(score, 7, n)]) for inc, score, n in groups])

kmeans = KMeans(n_clusters=5, n_init=10, random_state=42)
labels = kmeans.fit_predict(X)

print("Cluster sizes:", np.bincount(labels))
print("Centroids (income, score):")
print(kmeans.cluster_centers_.round(1))
print("Inertia (WCSS):", round(kmeans.inertia_, 1))
print("Iterations:", kmeans.n_iter_)

new_customers = np.array([[90, 85], [30, 25]])
print("New customers go to clusters:", kmeans.predict(new_customers))
```

OUTPUT

```
Cluster sizes: [50 40 30 40 40]
Centroids (income, score):
[[54.7 49.4]
 [85.5 80.2]
 [25.1 20.8]
 [85.6 20.4]
 [23.4 80.1]]
Inertia (WCSS): 17560.6
Iterations: 4
New customers go to clusters: [1 2]
```

The numerical example, step by step

PYTHON

```
import numpy as np

# 7 points, K = 2, starting centroids are A and D
names = ["A", "B", "C", "D", "E", "F", "G"]
X = np.array([[1, 1], [1.5, 2], [3, 4], [5, 7], [3.5, 5], [4.5, 5], [3.5, 4.5]])
centroids = np.array([[1, 1], [5, 7]], dtype=float)

for step in range(1, 10):
    dist = np.linalg.norm(X[:, None, :] - centroids[None, :, :], axis=2)   # distance to each centroid
    labels = dist.argmin(axis=1)
    print(f"Iteration {step}")
    for n, d, l in zip(names, dist.round(2), labels):
        print(f"  {n}: d1={d[0]:<5} d2={d[1]:<5} -> cluster {l + 1}")
    new = np.array([X[labels == k].mean(axis=0) for k in range(2)])
    print("  new centroids:", new.round(2).tolist())
    if np.allclose(new, centroids):
        print("Centroids did not move. Converged.")
        break
    centroids = new
```

OUTPUT

```
Iteration 1
  A: d1=0.0   d2=7.21  -> cluster 1
  B: d1=1.12  d2=6.1   -> cluster 1
  C: d1=3.61  d2=3.61  -> cluster 1
  D: d1=7.21  d2=0.0   -> cluster 2
  E: d1=4.72  d2=2.5   -> cluster 2
  F: d1=5.32  d2=2.06  -> cluster 2
  G: d1=4.3   d2=2.92  -> cluster 2
  new centroids: [[1.83, 2.33], [4.12, 5.38]]
Iteration 2
  A: d1=1.57  d2=5.38  -> cluster 1
  B: d1=0.47  d2=4.28  -> cluster 1
  C: d1=2.03  d2=1.78  -> cluster 2
  D: d1=5.64  d2=1.85  -> cluster 2
  E: d1=3.14  d2=0.73  -> cluster 2
  F: d1=3.77  d2=0.53  -> cluster 2
  G: d1=2.73  d2=1.08  -> cluster 2
  new centroids: [[1.25, 1.5], [3.9, 5.1]]
Iteration 3
  A: d1=0.56  d2=5.02  -> cluster 1
  B: d1=0.56  d2=3.92  -> cluster 1
  C: d1=3.05  d2=1.42  -> cluster 2
  D: d1=6.66  d2=2.2   -> cluster 2
  E: d1=4.16  d2=0.41  -> cluster 2
  F: d1=4.78  d2=0.61  -> cluster 2
  G: d1=3.75  d2=0.72  -> cluster 2
  new centroids: [[1.25, 1.5], [3.9, 5.1]]
Centroids did not move. Converged.
```

Elbow method and silhouette score

PYTHON

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

rng = np.random.default_rng(42)
groups = [(25, 20, 30), (25, 80, 40), (55, 50, 50), (85, 20, 40), (85, 80, 40)]
X = np.vstack([np.column_stack([rng.normal(inc, 7, n), rng.normal(score, 7, n)]) for inc, score, n in groups])

print(" K |   WCSS   | silhouette")
for k in range(1, 10):
    km = KMeans(n_clusters=k, n_init=10, random_state=42).fit(X)
    sil = silhouette_score(X, km.labels_) if k > 1 else float("nan")
    print(f"{k:2d} | {km.inertia_:8.0f} | {sil:.3f}")
```

OUTPUT

```
 K |   WCSS   | silhouette
 1 |   291347 | nan
 2 |   172512 | 0.391
 3 |   104377 | 0.502
 4 |    48400 | 0.622
 5 |    17561 | 0.706
 6 |    16047 | 0.597
 7 |    14511 | 0.549
 8 |    12864 | 0.475
 9 |    11521 | 0.400
```

Random initialisation vs k-means++

PYTHON

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, _ = make_blobs(n_samples=600, centers=8, cluster_std=0.6, random_state=11)

for init in ["random", "k-means++"]:
    inertias = [KMeans(n_clusters=8, init=init, n_init=1, random_state=s).fit(X).inertia_ for s in range(50)]
    inertias = np.array(inertias)
    best = inertias.min()
    print(f"{init:10s} best={best:7.1f}  worst={inertias.max():7.1f}  "
          f"runs within 1% of best: {np.mean(inertias < best * 1.01):.0%}")

km = KMeans(n_clusters=8, init="k-means++", n_init=10, random_state=0).fit(X)
print("k-means++ with n_init=10:", round(km.inertia_, 1))
```

OUTPUT

```
random     best=  395.0  worst= 5854.3  runs within 1% of best: 6%
k-means++  best=  394.7  worst=  443.4  runs within 1% of best: 58%
k-means++ with n_init=10: 394.7
```

Why feature scaling matters

PYTHON

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import adjusted_rand_score

# Two real groups of customers that differ by AGE; income is just noise
rng = np.random.default_rng(0)
age = np.concatenate([rng.normal(25, 3, 150), rng.normal(50, 3, 150)])
income = rng.normal(800000, 250000, 300)          # rupees per year
X = np.column_stack([age, income])
true_groups = np.repeat([0, 1], 150)

raw = KMeans(n_clusters=2, n_init=10, random_state=0).fit_predict(X)
scaled = KMeans(n_clusters=2, n_init=10, random_state=0).fit_predict(StandardScaler().fit_transform(X))

print("Agreement with the real groups (1.0 = perfect, 0 = random):")
print("  without scaling:", round(adjusted_rand_score(true_groups, raw), 3))
print("  with scaling:   ", round(adjusted_rand_score(true_groups, scaled), 3))
```

OUTPUT

```
Agreement with the real groups (1.0 = perfect, 0 = random):
  without scaling: -0.002
  with scaling:    1.0
```

K-means from scratch with NumPy

PYTHON

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

def kmeans(X, k, iters=100, seed=0):
    rng = np.random.default_rng(seed)
    centroids = X[rng.choice(len(X), k, replace=False)]           # k random points as start
    for i in range(iters):
        dist = ((X[:, None, :] - centroids[None, :, :]) ** 2).sum(axis=2)
        labels = dist.argmin(axis=1)                              # assignment step
        new = np.array([X[labels == j].mean(axis=0) if np.any(labels == j) else centroids[j]
                        for j in range(k)])                       # update step
        if np.allclose(new, centroids):
            break
        centroids = new
    inertia = ((X - centroids[labels]) ** 2).sum()
    return centroids, labels, inertia, i + 1

X, _ = make_blobs(n_samples=500, centers=4, cluster_std=0.8, random_state=3)

best = min((kmeans(X, 4, seed=s) for s in range(10)), key=lambda r: r[2])   # 10 restarts
c, labels, inertia, n_iter = best
sk = KMeans(n_clusters=4, n_init=10, random_state=0).fit(X)

print("Our inertia:         ", round(inertia, 2), "after", n_iter, "iterations")
print("scikit-learn inertia:", round(sk.inertia_, 2))
print("Our centroids:\n", c[np.lexsort(c.T)].round(2))
print("sklearn centroids:\n", sk.cluster_centers_[np.lexsort(sk.cluster_centers_.T)].round(2))
```

OUTPUT

```
Our inertia:          645.42 after 3 iterations
scikit-learn inertia: 645.42
Our centroids:
 [[-7.63 -5.71]
 [-4.12  0.15]
 [ 0.96  4.25]
 [ 7.77  8.05]]
sklearn centroids:
 [[-7.63 -5.71]
 [-4.12  0.15]
 [ 0.96  4.25]
 [ 7.77  8.05]]
```

Image compression (colour quantisation)

PYTHON

```
import numpy as np
from sklearn.cluster import KMeans
from sklearn.datasets import load_sample_image

image = load_sample_image("china.jpg")              # 427 x 640 photo that ships with scikit-learn
pixels = image.reshape(-1, 3) / 255.0               # every pixel is a point in RGB space
print("Pixels:", len(pixels), "| unique colours:", len(np.unique(image.reshape(-1, 3), axis=0)))

sample = pixels[np.random.default_rng(0).choice(len(pixels), 5000, replace=False)]
for k in [4, 8, 16]:
    km = KMeans(n_clusters=k, n_init=4, random_state=0).fit(sample)
    labels = km.predict(pixels)
    compressed = km.cluster_centers_[labels].reshape(image.shape)   # repaint each pixel with its centroid
    error = np.abs(compressed - image / 255.0).mean() * 255
    print(f"K={k:2d}: {k} colours, average error {error:.1f} out of 255")
```

OUTPUT

```
Pixels: 273280 | unique colours: 96615
K= 4: 4 colours, average error 17.4 out of 255
K= 8: 8 colours, average error 11.2 out of 255
K=16: 16 colours, average error 8.0 out of 255
```

11. K-means interview questions

1. What is k-means clustering?

An unsupervised algorithm that partitions data into K clusters. It places K centroids and assigns each point to the nearest one, then moves each centroid to the mean of its points, repeating until the clusters stop changing.

2. Explain the steps of the k-means algorithm.

Choose K, initialise K centroids, assign each point to its nearest centroid, update each centroid to the mean of its assigned points, and repeat the assign and update steps until assignments stop changing or a maximum number of iterations is reached.

3. What objective does k-means minimise?

The within-cluster sum of squares (WCSS), also called inertia: the sum of squared Euclidean distances from each point to its cluster centroid.

4. Does k-means always converge? Does it find the global optimum?

It always converges, because neither step can increase WCSS and the number of possible partitions is finite. But it only reaches a local minimum, which depends on the starting centroids.

5. How do you choose the value of K?

Use the elbow method on a plot of WCSS against K, the silhouette score, or the gap statistic, together with domain knowledge about how many groups are useful.

6. What is the elbow method?

You plot WCSS for a range of K values and pick the K at the bend of the curve, after which adding clusters gives only small reductions in WCSS.

7. What is the silhouette score?

For each point, $s = (b - a) / \max(a, b)$, where a is the mean distance to points in its own cluster and b is the mean distance to points in the nearest other cluster. The average ranges from -1 to 1; higher means better separated clusters.

8. What is k-means++?

An initialisation method that picks the first centroid at random and each next centroid with probability proportional to its squared distance from the nearest centroid already chosen. It spreads the centroids out and gives better, more consistent results.

9. Why should you scale features before k-means?

Because k-means uses Euclidean distance, features with larger ranges dominate the distance. Standardising puts all features on a comparable scale.

10. What are the main limitations of k-means?

You must choose K, it assumes convex clusters of similar size and density, it is sensitive to initialisation and outliers, it needs numeric scaled features, and it assigns every point to some cluster.

11. How does k-means handle outliers?

Poorly. Outliers pull centroids towards them because the centroid is a mean. You can remove outliers first, or use K-medoids or DBSCAN.

12. What is the difference between k-means and KNN?

K-means is unsupervised clustering where K is the number of clusters. KNN is supervised classification or regression where K is the number of neighbours used to predict a label.

13. What is the difference between k-means and k-medoids?

K-means uses the mean of the points as the centre, which may not be an actual data point. K-medoids uses an actual data point (the medoid) and can work with any distance, which makes it more robust to outliers.

14. What is the time complexity of k-means?

About O(n × K × d × i) for n points, K clusters, d features and i iterations. It is linear in the number of points, which is why it scales well.

15. What happens if a cluster becomes empty?

Its centroid has no points to average. Implementations handle it differently: keep the old centroid, or move it to a point far from the others, as scikit-learn does.

16. What is mini-batch k-means?

A variant that updates the centroids using small random batches of data instead of the whole dataset each round. It is much faster on large data, with slightly worse inertia.

17. Can k-means be used for categorical data?

Not directly, because means and Euclidean distances do not make sense for categories. Use k-modes for categorical data or k-prototypes for mixed data.

18. How can k-means help a supervised model?

The cluster label, or the distance to each centroid, can be added as a feature. K-means can also be used to compress data or choose representative samples.

Watch k-means step by step in the interactive playground, try the elbow and silhouette tool and take the quiz on the blog: codeanddebug.in/blog/k-means-clustering