2025-11-16 13:44

tags :

- Machine Learning
- Hands on ML - book

Chapter 9 - Hands on

1. Core Concept & Importance
   - **Definition:** Unsupervised learning works with **unlabeled data**, where you have input features (X) but no corresponding labels (y).
   - **Significance:** While most current ML applications use supervised learning, the vast majority of available data is unlabeled.
   - **Yann LeCun's Analogy:** Unsupervised learning is the "cake" of intelligence, supervised learning is the "icing," and reinforcement learning is the "cherry," highlighting its foundational importance.
2. Key Unsupervised Learning Tasks Covered in this Chapter
   - **Clustering:**
     - **Goal:** To group similar instances together into clusters.
     - **Applications:** Customer segmentation, recommender systems, image segmentation, data analysis.
   - **Anomaly Detection (Outlier Detection):**
     - **Goal:** To learn what "normal" data (inliers) looks like in order to identify abnormal instances (anomalies or outliers).
     - **Applications:** Fraud detection, finding defective products, removing outliers to improve other models.
   - **Density Estimation:**
     - **Goal:** To estimate the Probability Density Function (PDF) of the data.
     - **Applications:** Commonly used for anomaly detection (instances in low-density areas are likely anomalies), data analysis, and visualization.
3. Chapter Roadmap: Algorithms to be Discussed
   - **Clustering Algorithms:**
     - k-Means
     - DBSCAN
   - **Versatile Model:**

- **Gaussian Mixture Models:** Can be used for density estimation, clustering, and anomaly detection.

Clustering Algorithms: k-means and DBSCAN

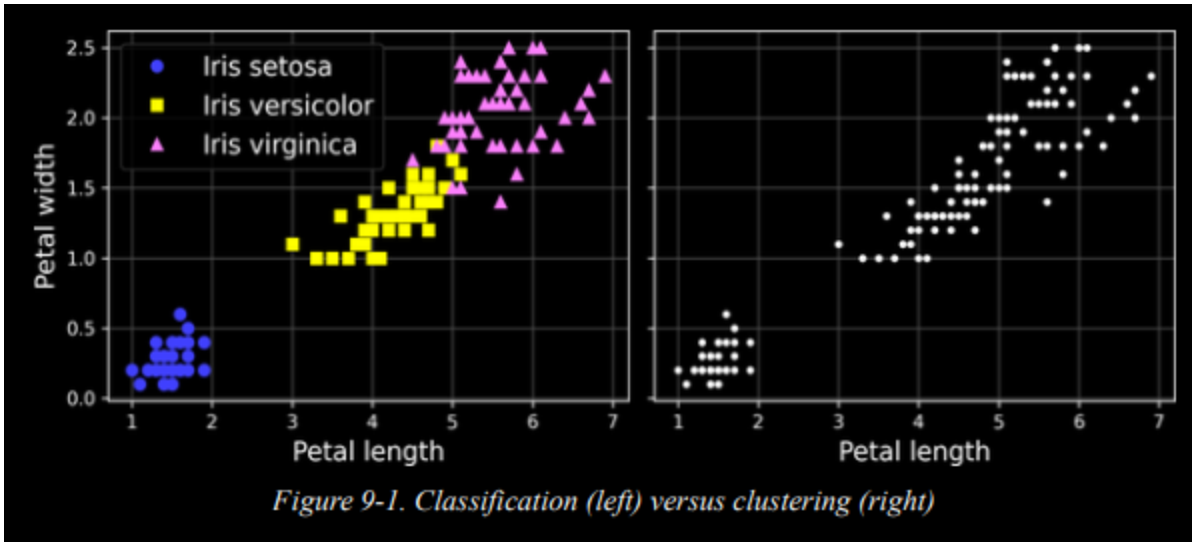
- Clustering vs. Classification
  This is the main concept shown in Figure 9-1.
  - **Classification (Left Image):**
    - This is a **supervised** task.
    - The data is **labeled** (we know the species: Setosa, Versicolor, Virginica).
    - The goal is to train a model to assign these known labels to new data.
  - **Clustering (Right Image):**
    - This is an **unsupervised** task.
    - The data is **unlabeled** (all points look the same; we just have measurements).
    - The goal is to *discover* hidden groups in the data.
- *Classification (Left Image), Clustering (Right Image)*
  - 

    Figure 9-1. Classification (left) versus clustering (right)

k-means

- Core Algorithm Concept
  - **Purpose**: Quickly clusters data into clearly separated blobs (like the 5-blob example in Figure 9-2)

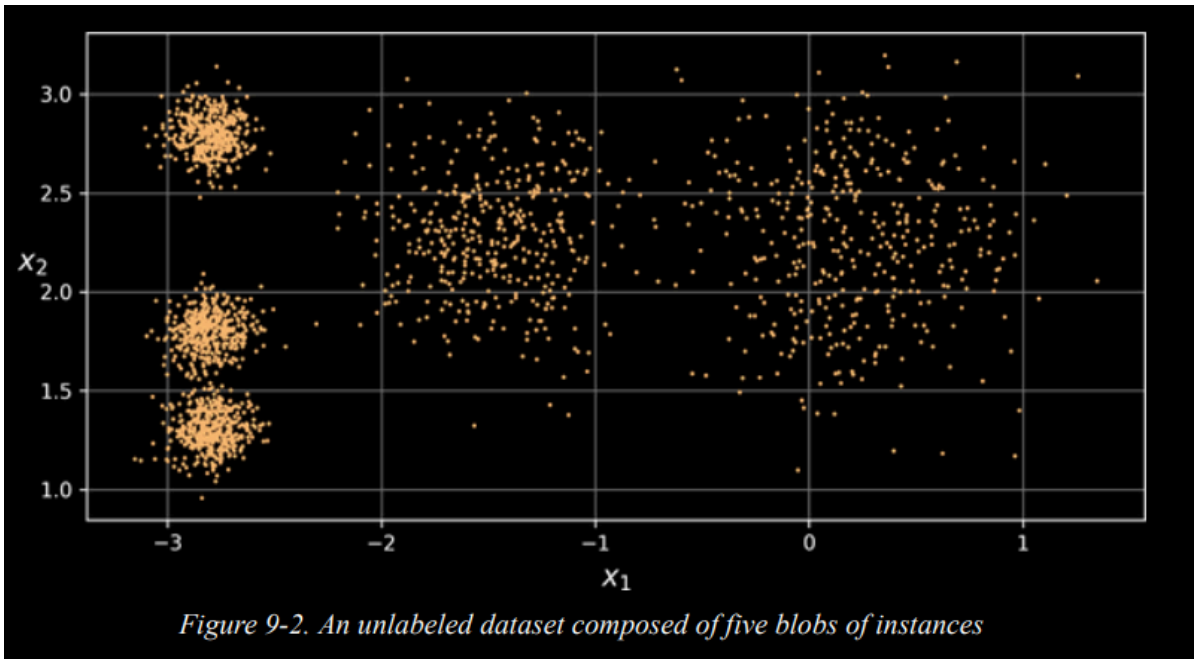
- 

Figure 9-2. An unlabeled dataset composed of five blobs of instances

- Key Implementation Details
  - **Training Process:**

```
from sklearn.cluster import KMeans
k = 5  # Must specify number of clusters
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

    - **Must specify k in advance** - obvious in the 5-blob example, but generally challenging in real applications
  - **Output Properties:**
    - `y_pred` contains cluster indices (0 to 4 for k=5) - these are *predicted cluster labels*, not true class labels
    - `y_pred is kmeans.labels_` returns `True` - both refer to the same cluster assignments
    - `kmeans.cluster_centers_` gives coordinates of the five centroids
- Prediction on New Data

```
X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
kmeans.predict(X_new)  # Returns array([1, 1, 2, 2])
```

  - New instances assigned to cluster with closest centroid
- K-Means Short Notes
  - **Problems**
    - Fails with different cluster sizes
    - Bad at boundary instances
    - Only does **hard clustering** (1 cluster per point)
  - **Solution: Soft Clustering**

    - Get **distance to each centroid** instead of just one cluster
    - Use `kmeans.transform(X)` → returns distance matrix
  - **Uses**
    1. **Better features**: Use distances as input for other models
    2. **Dimensionality reduction**: Convert to k-distance features
  - **Example:**

```
distances = kmeans.transform(X_new)
# Returns: [[2.81, 0.33, 2.90, ...]]
# Distances to all centroids, not just one
```

The k-means algorithm

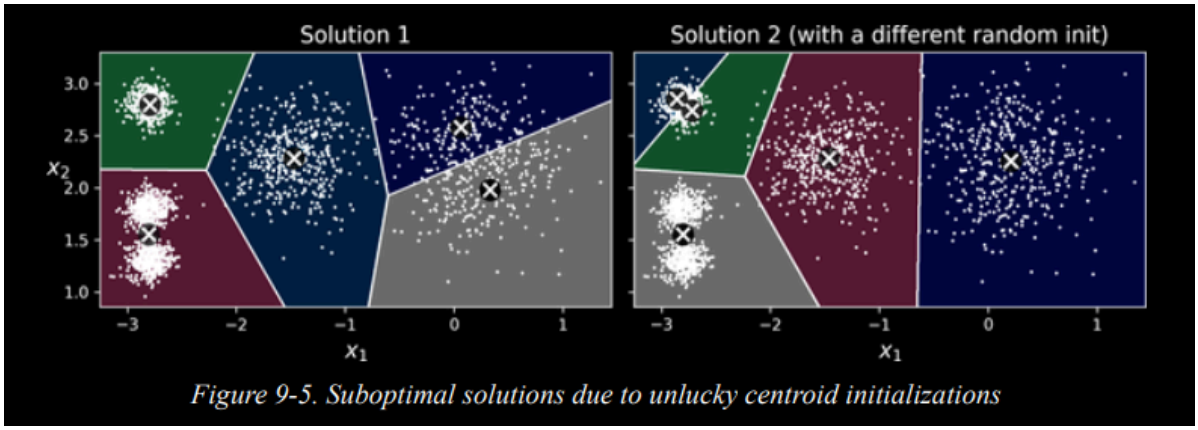
- The Core Process
  1. **Start**: Place `k` centroids randomly
  2. **Label**: Assign each instance to closest centroid
  3. **Update**: Move centroids to mean of their instances
  4. **Repeat** steps 2-3 until centroids stop moving
- Key Properties
  - **Guaranteed convergence** in few steps
  - **Fast**: Complexity ≈ O(m × k × n) - linear with instances, clusters, dimensions
  - **But**: May converge to **local optimum** (bad solution)
- The Problem: Random Initialization
  - Different random starts → different solutions
  - Figure 9-5 shows **suboptimal solutions** from bad initialization
  - 

    Figure 9-5. Suboptimal solutions due to unlucky centroid initializations
  - Need better initialization methods
- Visual Flow:
  Random centroids → Label instances → Update centroids → Relabel → Converge

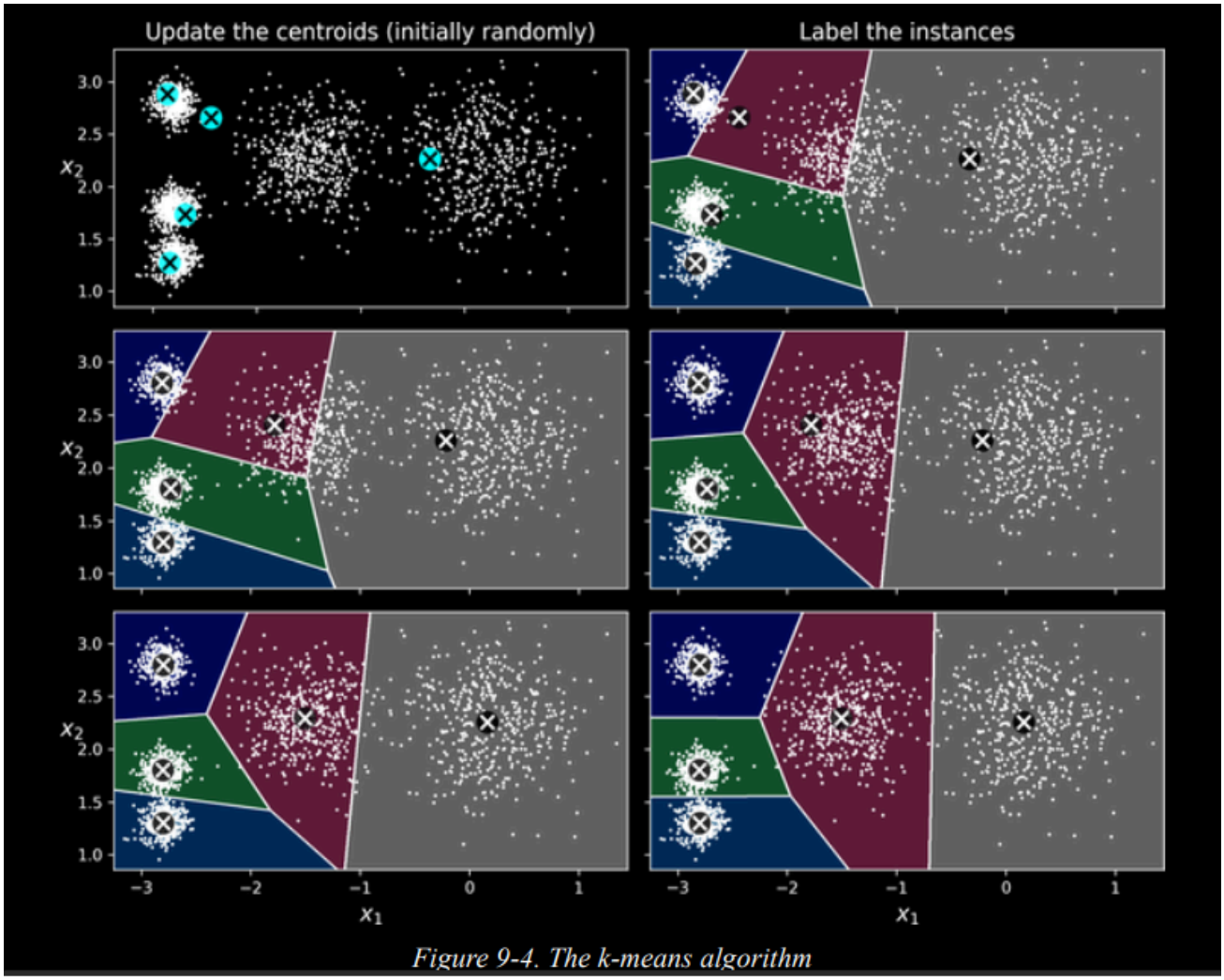


Figure 9-4. The k-means algorithm

Centroid initialization methods

- Problem
  - Random initialization can lead to bad local optima
  - Need better ways to start centroids
- Solutions
  1. Manual Initialization

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(init=good_init, n_init=1)
```

     - Use when you know approximate centroid positions
  2. Multiple Runs (Default)
     - `n_init=10` - runs algorithm 10 times with different random starts
     - Keeps **best solution** based on **inertia**
     - Inertia = Performance Metric
       - **Definition**: Sum of squared distances from instances to closest centroids
       - **Lower inertia = better clustering**

- Access with `kmeans.inertia_`
- `score()` returns **negative inertia** (to follow "greater is better" rule)

3. K-Means++ (Smart Default)
   - **Smarter initialization** that spreads centroids apart
   - **Process**:
     1. Pick first centroid randomly
     2. Pick next centroid with probability proportional to distance² from existing centroids
     3. Repeat until k centroids
   - **Result**: Fewer runs needed, better solutions
   - **Key Point**: K-means++ is now the **default** in sklearn - much better than pure random

Accelerated k-means and mini-batch k-means

1. Accelerated K-Means (Elkan's)
   - **Idea**: Uses triangle inequality to avoid unnecessary distance calculations
   - **Result**: Sometimes faster, sometimes slower - depends on dataset
   - **Use**: `algorithm="elkan"` in KMeans
2. Mini-Batch K-Means
   - **Idea**: Uses small batches of data instead of full dataset each iteration
   - **Speed**: **3-4x faster** than regular k-means
   - **Trade-off**: Slightly **worse inertia** (lower quality clusters)

```
from sklearn.cluster import MiniBatchKMeans
minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

- For Huge Datasets
  - *Option 1:* Use `memmap` (like Chapter 8 PCA)
  - *Option 2:* Use `partial_fit()` manually (more work)
- Key Trade-off
  - **Regular K-means**: Better quality, slower
  - **Mini-batch**: Worse quality, much faster
  - **Choose based on**: Dataset size vs. quality needs
  - **Figure 9-6 shows**: Inertia slightly worse but speed much better, especially with more clusters

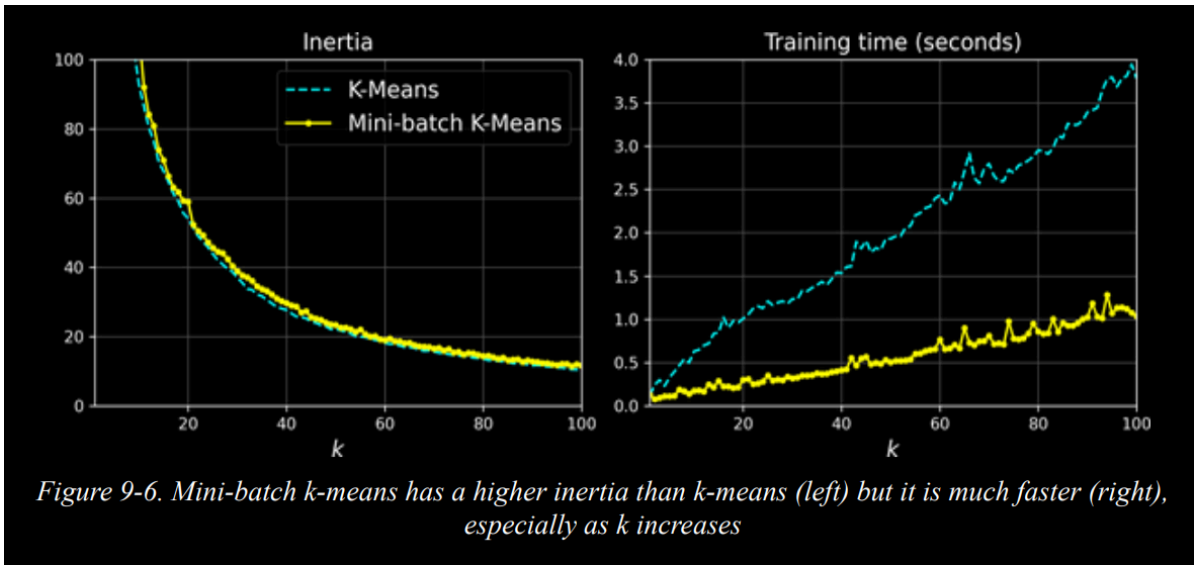
- 

Figure 9-6. Mini-batch k-means has a higher inertia than k-means (left) but it is much faster (right), especially as k increases

Finding the optimal number of clusters

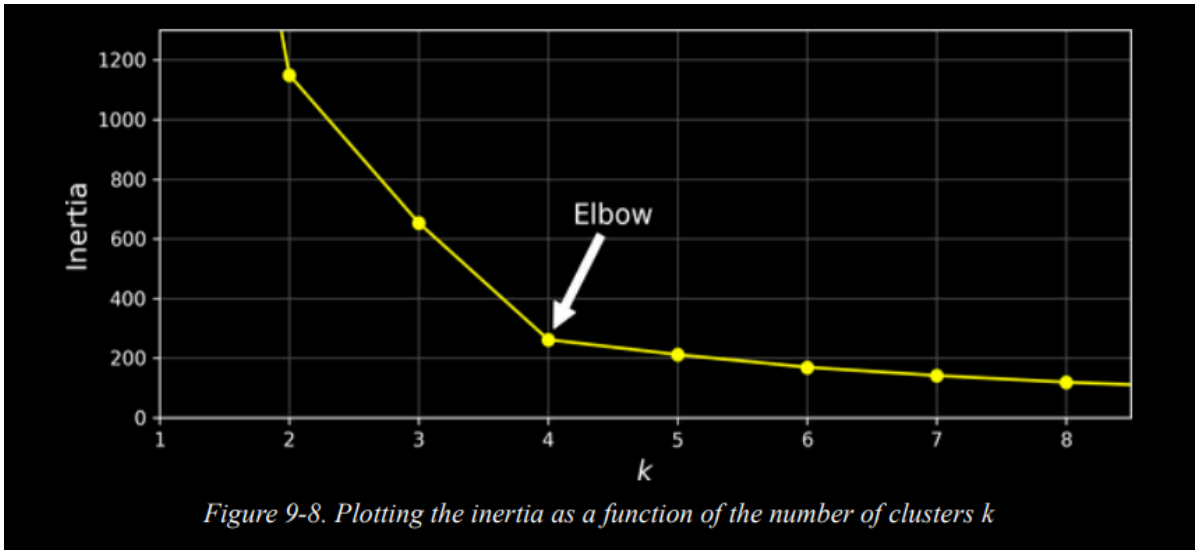
- The Problem with Inertia
  - **Inertia always decreases** as k increases
  - Can't pick k by minimizing inertia
  - **Elbow method**: Find where inertia curve bends. it will be more appropriate to choose the inflexion point (elbow)
    - 

Figure 9-8. Plotting the inertia as a function of the number of clusters k

- *Better Method: Silhouette Analysis*
  - Silhouette Coefficient Formula:

```
(b - a) / max(a, b)
```

    - **a** = mean distance to other points in same cluster
    - **b** = mean distance to points in nearest other cluster
    - **Range**: -1 to +1
      - **+1**: Perfectly inside own cluster

    - `0`: On cluster boundary
    - `-1`: Probably wrong cluster
- Silhouette Score
    - Mean silhouette coefficient for all instances
    - Higher score = better clustering

```
from sklearn.metrics import silhouette_score
silhouette_score(X, kmeans.labels_)
```

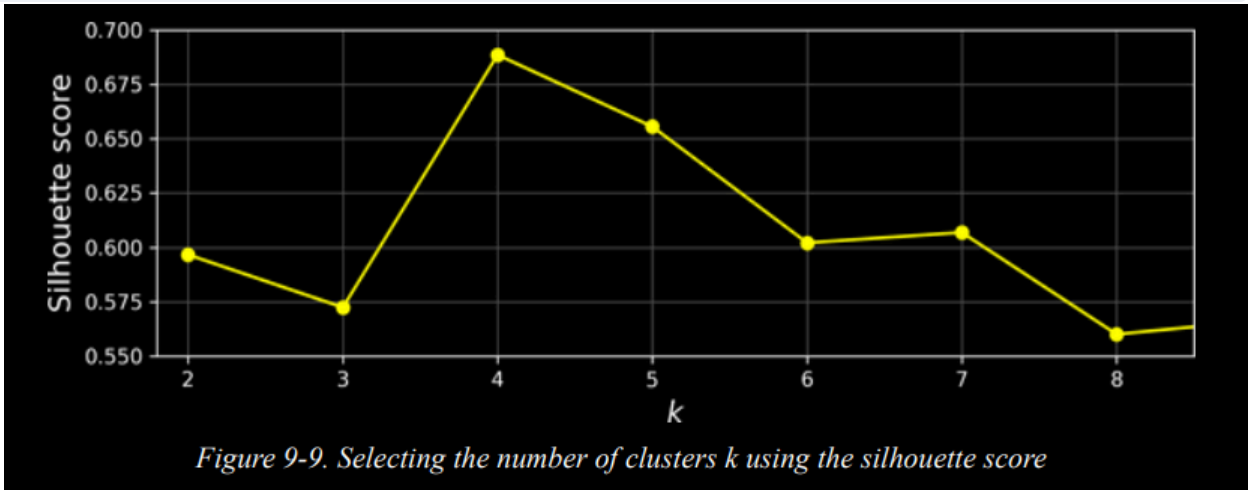
    - 

Figure 9-9. Selecting the number of clusters k using the silhouette score
    - the best nuber is the highest score is `4`
- Silhouette Diagrams

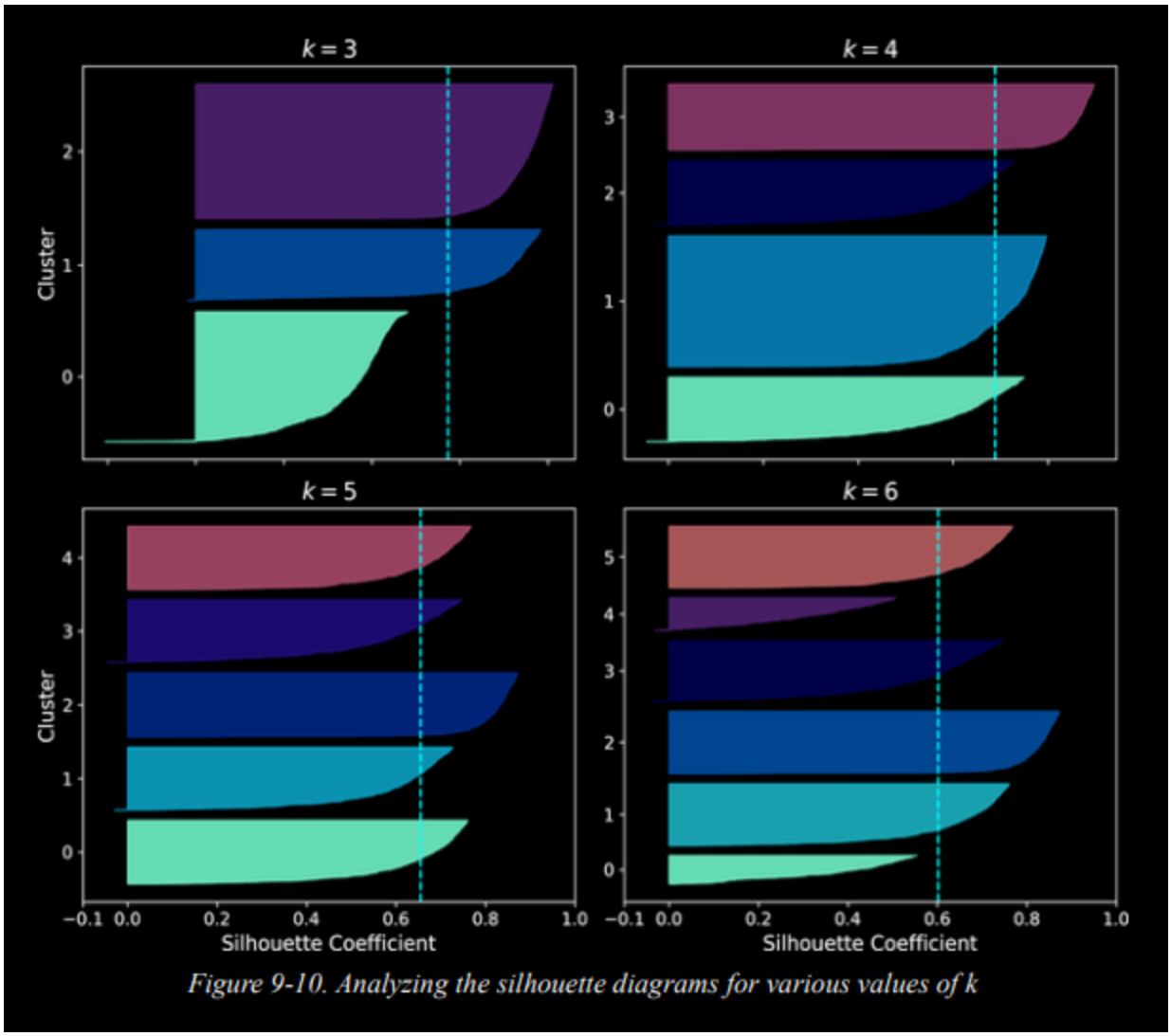


Figure 9-10. Analyzing the silhouette diagrams for various values of k

-
  - Each "knife" = one cluster
  - Height = number of instances in cluster
  - Width = silhouette coefficients (wider = better)
  - Dashed line = mean silhouette score
- How to Read Diagrams
  - Good: Most instances extend right of dashed line
  - Bad: Many instances stop left of dashed line
  - Good k: All clusters have good width, similar sizes
- Example Analysis
  - k=4 & k=5: Both good (wide knives, past dashed line)
  - k=5: Better because clusters more balanced in size
  - k=3 & k=6: Bad (narrow knives, many instances left of line)
    Best Approach: Use silhouette diagrams, not just scores!

Limits of k-means

- Main Problems

1. **Need multiple runs** to avoid bad solutions
2. **Must specify k** in advance
3. **Fails with complex cluster shapes**

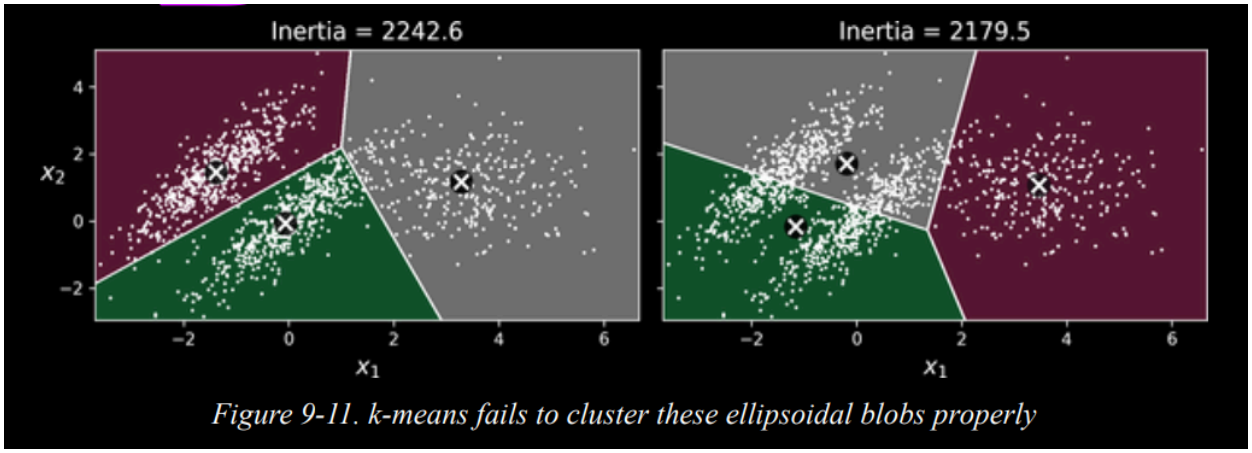
- Where K-Means Fails
  - 

Figure 9-11. k-means fails to cluster these ellipsoidal blobs properly
  - **Varying sizes** - different cluster diameters
  - **Different densities** - some clusters packed tight, others spread out
  - **Non-spherical shapes** - elliptical, elongated clusters
  - **Example**: Ellipsoidal clusters get chopped incorrectly
- Key Insight
  - **Lower inertia ≠ better clustering** (right solution in figure has lower inertia but is terrible)
  - K-means assumes clusters are spherical and similar size
- Solutions
  1. **Scale features first** - helps but doesn't fix everything
  2. **Use other algorithms** for complex shapes:
     - **Gaussian Mixture Models** work well for elliptical clusters
     - **DBSCAN** for varying densities
- When to Use K-Means
  - Fast and scalable
  - When clusters are roughly spherical and similar size
  - Good baseline algorithm

  Remember: Always check if your cluster shapes match k-means assumptions!

Using Clustering for Image Segmentation

- Types of Image Segmentation
  1. *Color Segmentation:* Group pixels by similar color
  2. *Semantic Segmentation:* Group pixels by object type (e.g., all pedestrians)

3. *Instance Segmentation:* Group pixels by individual objects (e.g., each pedestrian separately)

   K-means is good for color segmentation, CNNs are better for semantic/instance

- How K-Means Color Segmentation Works
  - *Step 1: Prepare Data*

```
image = np.asarray(PIL.Image.open(filepath))  # Shape: (height,
width, 3)
X = image.reshape(-1, 3)  # Flatten to list of RGB pixels
```

  - **Step 2: Cluster Colors**

```
kmeans = KMeans(n_clusters=8).fit(X)
```

  - **Step 3: Recreate Image**

```
segmented_img = kmeans.cluster_centers_[kmeans.labels_]
segmented_img = segmented_img.reshape(image.shape)
```

- Results

Figure 9-12. Image segmentation using k-means with various numbers of color clusters

  - **Fewer clusters** = more compression, less detail
  - **More clusters** = better color preservation
  - **Problem**: Small but important colors (ladybug red) may get lost with few clusters

Using Clustering for Semi-Supervised Learning

- The Problem
  - Only **50 labeled images** out of 1,797 digits
  - Baseline logistic regression: **74.8% accuracy**
  - Full training set gets ~90.7% - how to bridge the gap?

```
from sklearn.datasets import load_digits
from sklearn.linear_model import LogisticRegression

# Load and split data
X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]

# Train on only 50 random labels
n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
log_reg.score(X_test, y_test)  # 74.8%
```

Step 1: Smart Labeling with Clustering

1. Cluster training set into **50 clusters** (k=50)
2. Find **representative image** closest to each centroid
3. **Manually label** only these 50 representatives

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

   # Cluster and find representative images
   k = 50
   kmeans = KMeans(n_clusters=k, random_state=42)
   X_digits_dist = kmeans.fit_transform(X_train)  # Distance to each
centroid
   representative_digit_idx = np.argmin(X_digits_dist, axis=0)  # Closest to
centroid
   X_representative_digits = X_train[representative_digit_idx]

   # Manually label the 50 representatives
   y_representative_digits = np.array([1, 3, 6, 0, 7, 9, 2, 4, 8, 9,
                                       5, 4, 7, 1, 2, 6, 1, 2, 5, 1,
                                       4, 1, 3, 3, 8, 8, 2, 5, 6, 9,
                                       1, 4, 0, 6, 8, 3, 4, 6, 7, 2,
                                       4, 1, 0, 7, 5, 1, 3, 4, 3, 7])

   # Train on smart labels
   log_reg = LogisticRegression(max_iter=10_000)
   log_reg.fit(X_representative_digits, y_representative_digits)
   log_reg.score(X_test, y_test)  # 84.9% (vs 74.8% with random labels)
```

Step 2: Label Propagation

- Assign each cluster's label to **all instances** in that cluster
- Now train on **entire training set** with propagated labels

```
# Propagate labels to entire cluster
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]

# Train on all propagated labels
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train, y_train_propagated)
log_reg.score(X_test, y_test)  # 89.4%
```

Step 3: Remove Outliers

- Remove 1% farthest instances from cluster centers (likely mislabeled)
- Train on cleaned dataset

```
# Remove 1% farthest from centroids
percentile_closest = 99
X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]

for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]

# Train on cleaned data
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
log_reg.score(X_test, y_test)  # 90.9%
```

- there is alternative way

```
from sklearn.semi_supervised import LabelPropagation,
SelfTrainingClassifier

# Automatic label propagation
label_prop = LabelPropagation()
label_prop.fit(X_train, y_train_propagated)  # Uses similarity matrix

# Self-training
from sklearn.ensemble import RandomForestClassifier
self_training = SelfTrainingClassifier(RandomForestClassifier())
self_training.fit(X_train, y_train_initial)
```

DBSCAN

Resources :

-

Related notes :

-

References :

- Internal :
  -
  -
  -
- External :
  - hegab videos
  - the book
  - the notebook
  -