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

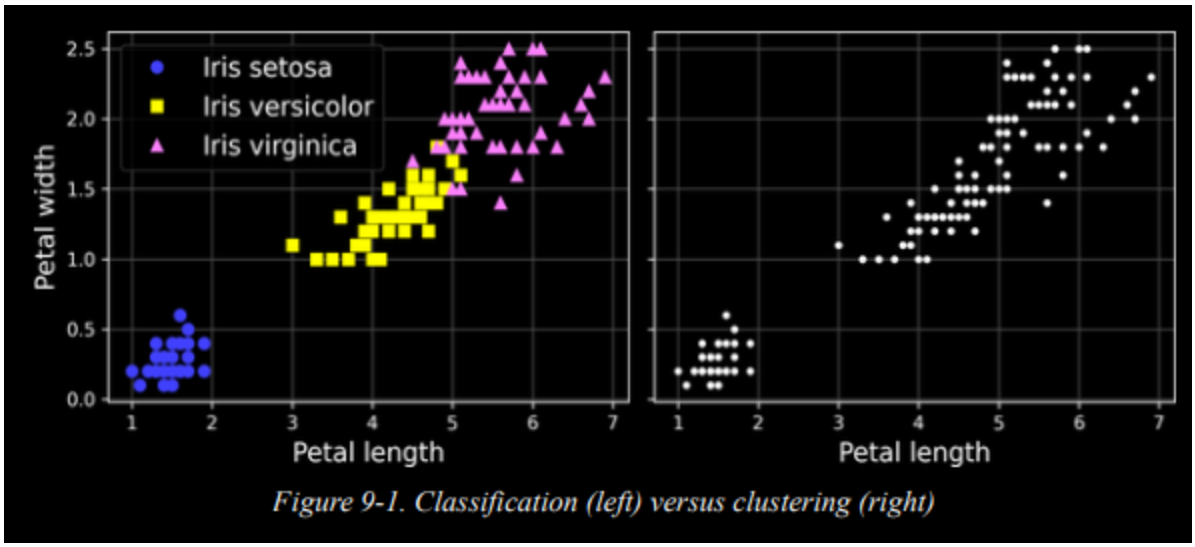
- Clustering vs. Classification
  This is the main concept shown in Figure 9-1.
  - **Classification (Left Image):**
    - This is a **supervised** task.
    - The data is **labeled** (we know the species: Setosa, Versicolor, Virginica).
    - The goal is to train a model to assign these known labels 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)

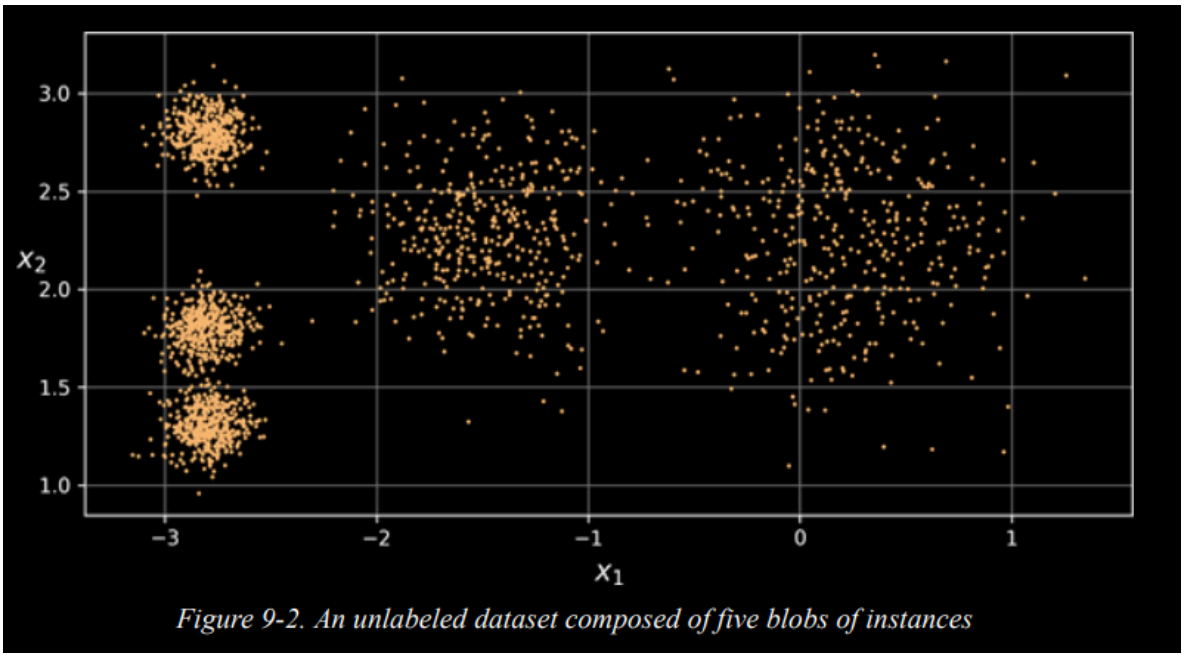
- 

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

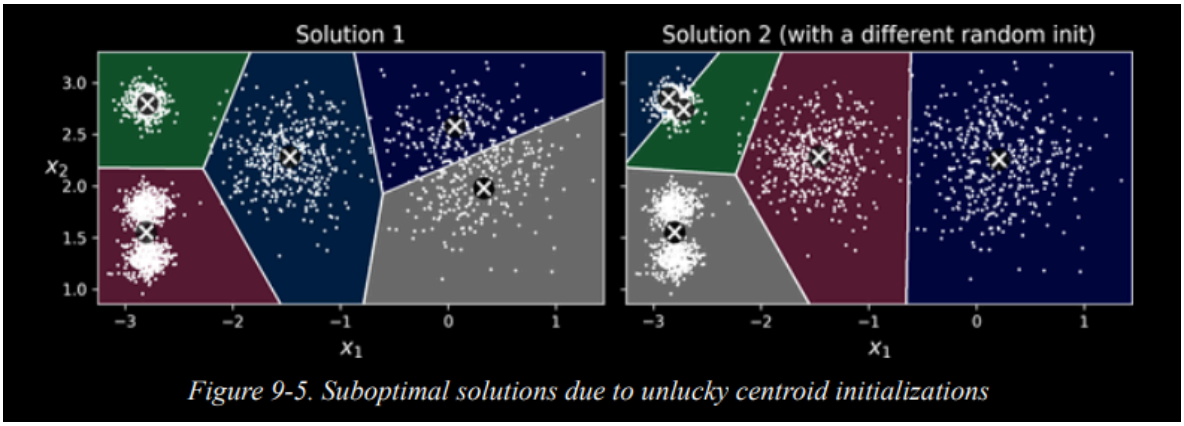
- The Core Process
  1. **Start**: Place `k` centroids randomly
  2. **Label**: Assign each instance to closest centroid
  3. **Update**: Move centroids to mean of their instances
  4. **Repeat** steps 2-3 until centroids stop moving
- Key Properties
  - **Guaranteed convergence** in few steps
  - **Fast**: 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

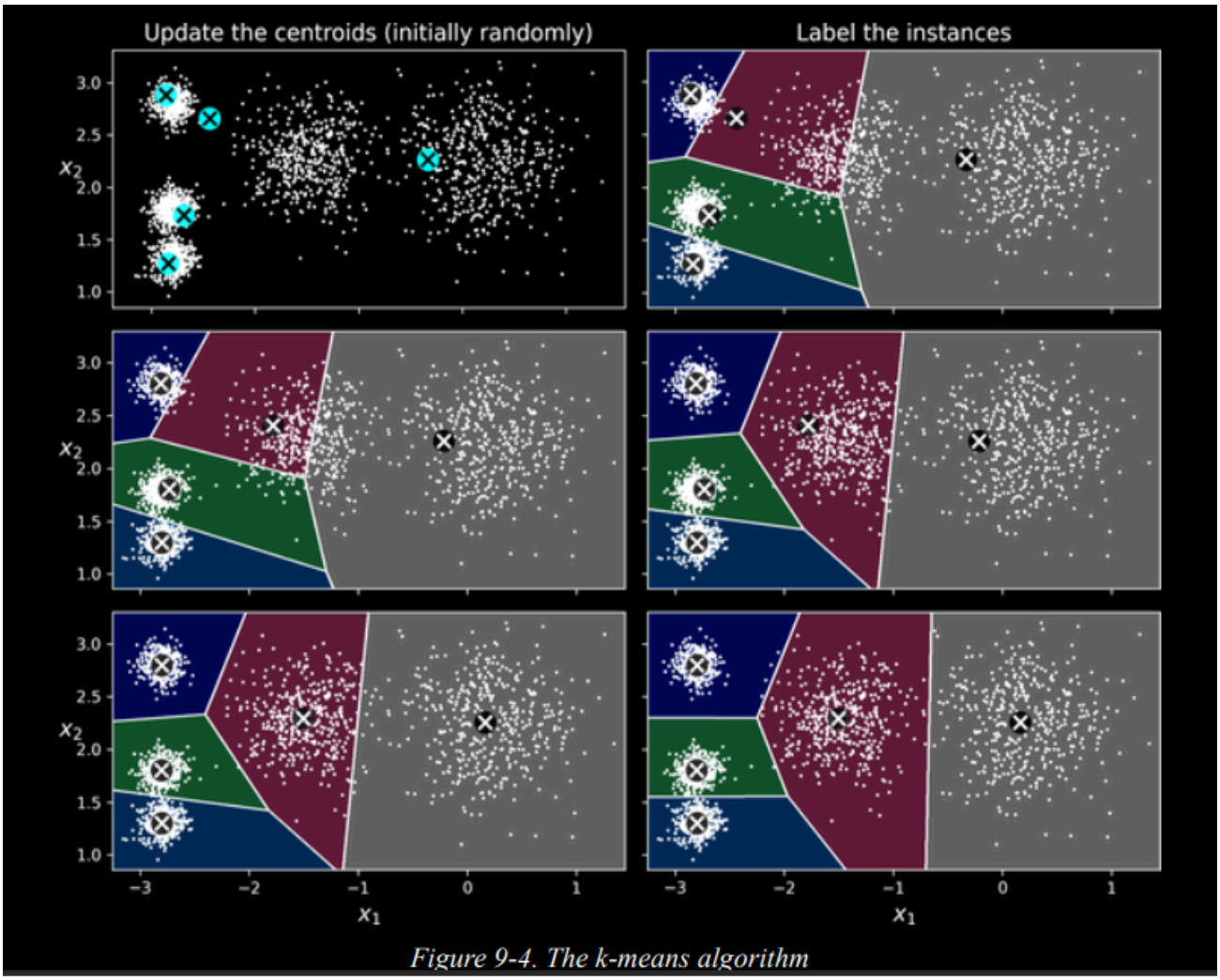


Figure 9-4. The k-means algorithm

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

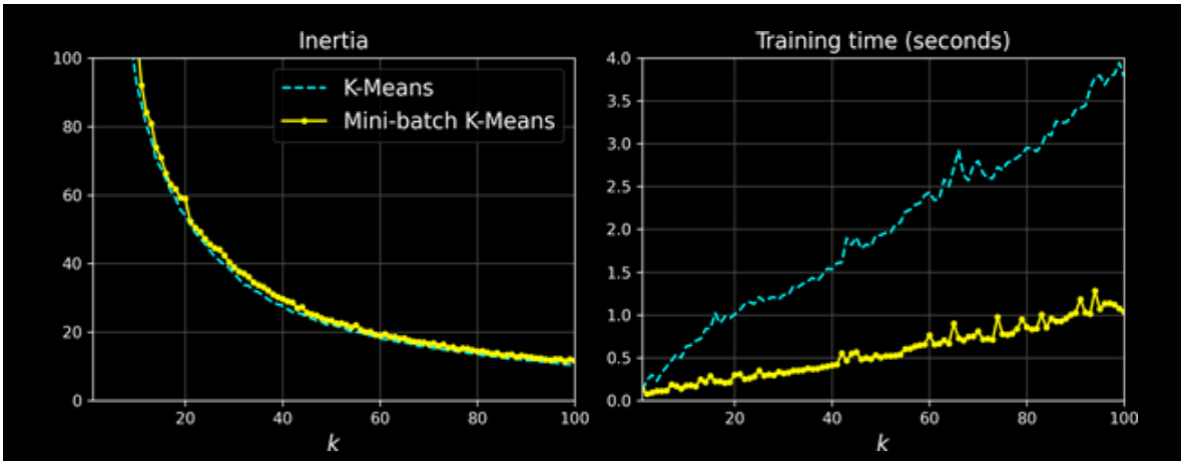
- **Choose based on**: Dataset size vs. quality needs
- **Figure 9-6 shows**: Inertia slightly worse but speed much better, especially with more clusters
  - 

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

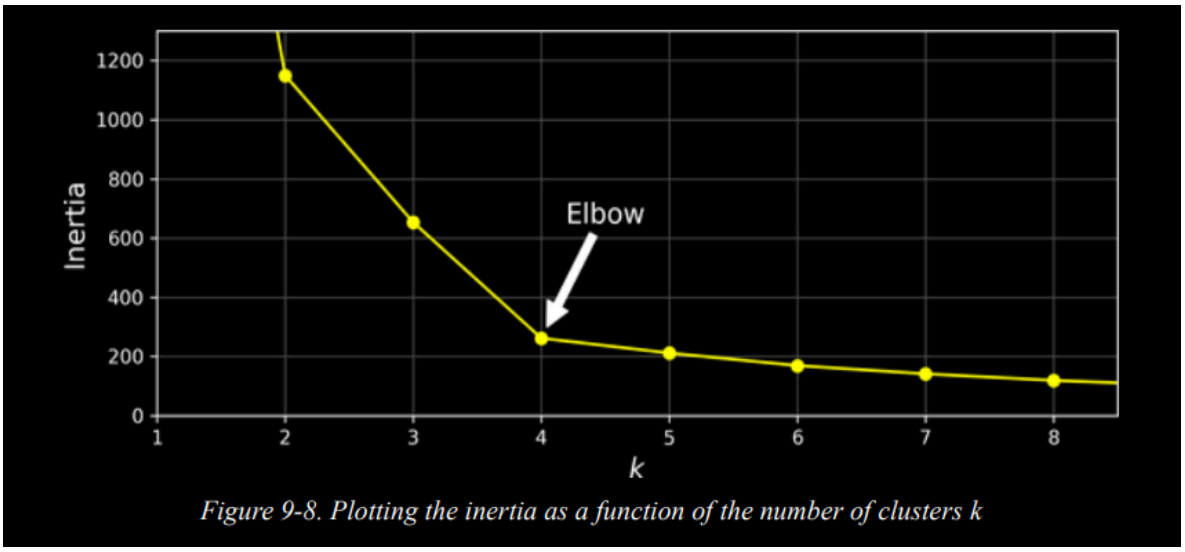
- The Problem with Inertia
  - **Inertia always decreases** as k increases
  - Can't pick k by minimizing inertia
  - **Elbow method**: Find where inertia curve bends. it will be more appropriate to choose the inflexion point (elbow)
    - 

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_)
```

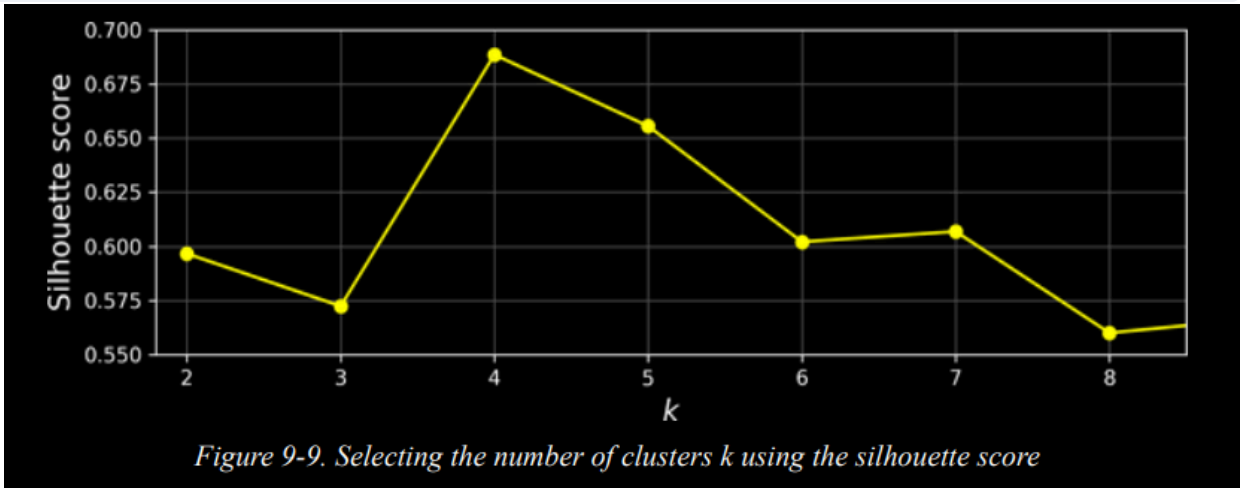


Figure 9-9. Selecting the number of clusters k using the silhouette score

    - the best nuber is the highest score is `4`
- Silhouette Diagrams

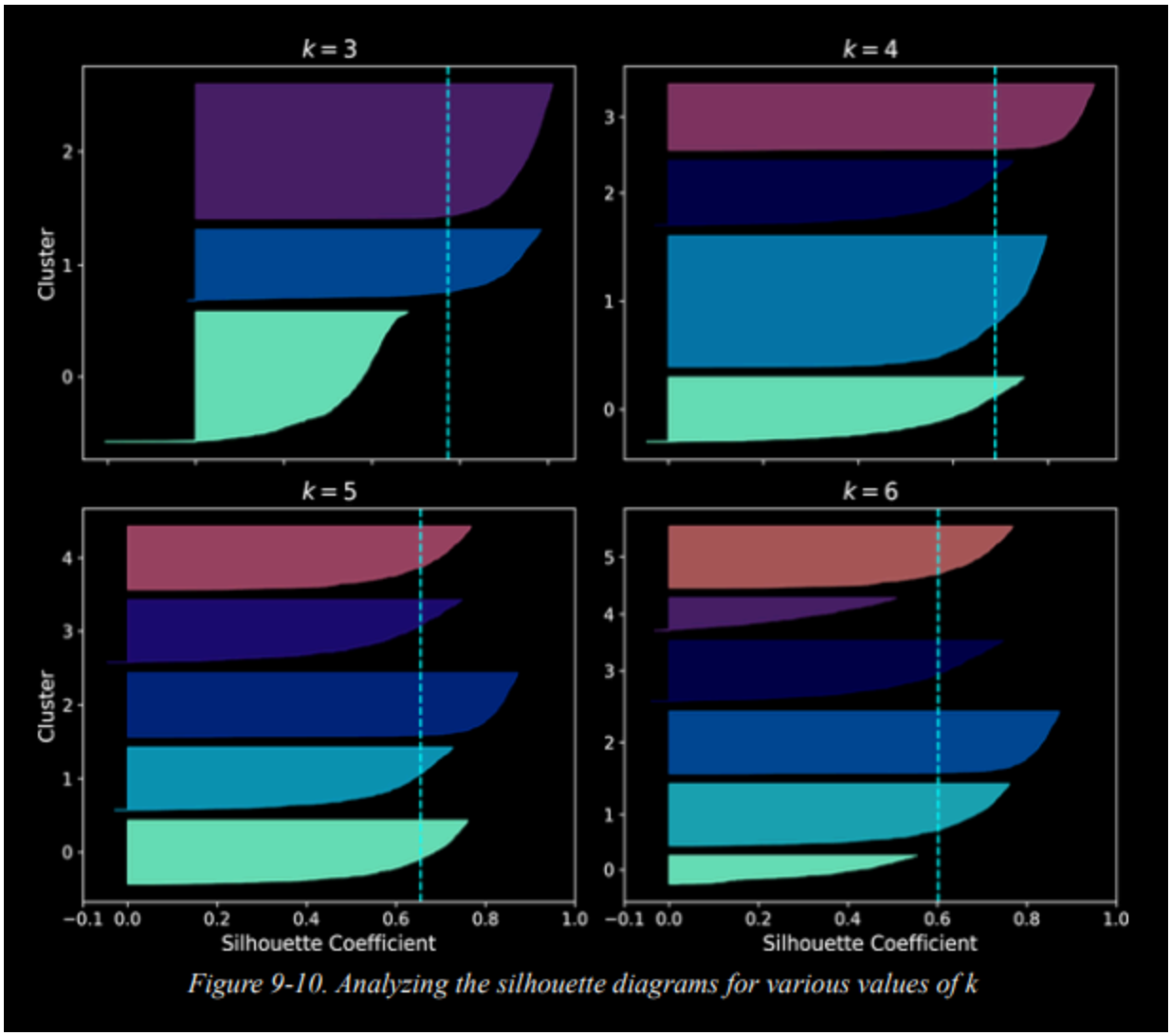


Figure 9-10. Analyzing the silhouette diagrams for various values of k

-
  - 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

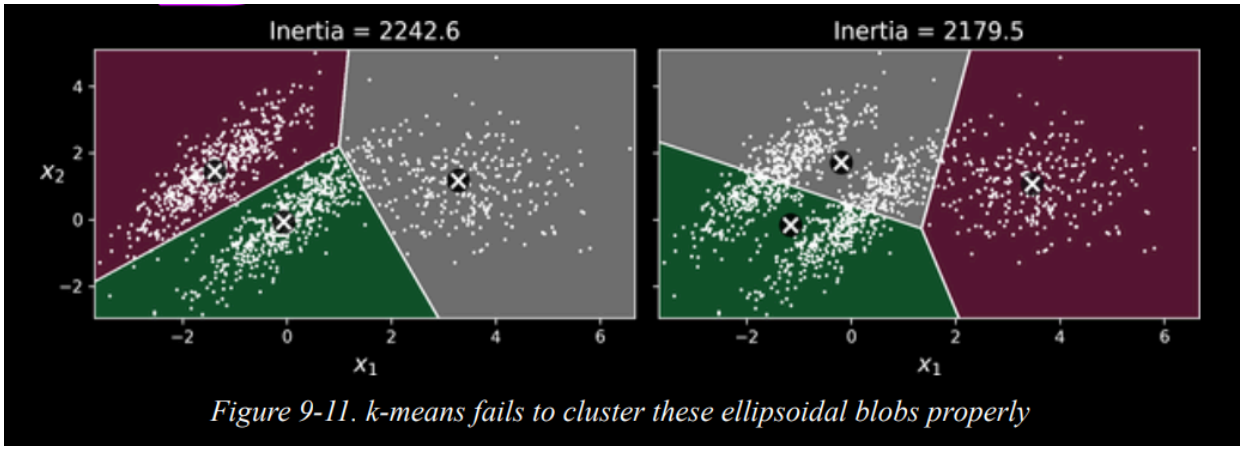


Figure 9-11. k-means fails to cluster these ellipsoidal blobs properly

  - **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

- How DBSCAN Works
  1. **ε-neighborhood**: For each point, count instances within distance `ε`
  2. **Core instance**: Has ≥ `min_samples` in its neighborhood
  3. **Cluster**: All points reachable from core instances
  4. **Anomalies**: Points that aren't core instances and have no core in neighborhood
- Code Implementation

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05)
dbscan = DBSCAN(eps=0.2, min_samples=5)
dbscan.fit(X)
```

  - Key Outputs

```
dbscan.labels_           # Cluster labels (-1 = anomaly)
dbscan.core_sample_indices_  # Indices of core instances
dbscan.components_       # The core instances themselves
```

- Hyperparameter Tuning

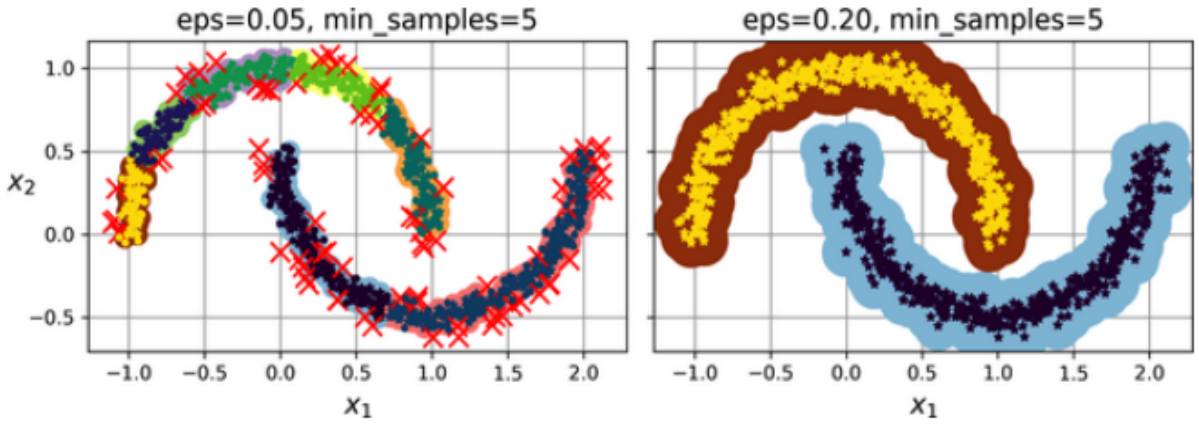


Figure 9-14. DBSCAN clustering using two different neighborhood radiuses

- `eps=0.05`: Too small → many small clusters & anomalies
- `eps=0.2`: Good → finds the two moon shapes perfectly

- Predicting New Instances
  DBSCAN has no `predict()` method, so we use KNN:

```
from sklearn.neighbors import KNeighborsClassifier

# Train on core instances only
knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_,
        dbscan.labels_[dbscan.core_sample_indices_])

# Predict new instances
X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
knn.predict(X_new)        # Cluster assignments
knn.predict_proba(X_new)  # Probabilities
```

  - Anomaly Detection for New Data

```
# Find distance to nearest neighbor
y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
y_pred = dbscan.labels_[dbscan.core_sample_indices_][y_pred_idx]

# Mark far points as anomalies
y_pred[y_dist > 0.2] = -1
y_pred.ravel()  # array([-1, 0, 1, -1])
```

- Pros & Cons
  - Advantages:
    - Finds clusters of any shape
    - Robust to outliers
    - Only 2 hyperparameters
  - Limitations:

- Struggles with varying densities
- Computational complexity: O(m × n) - slow for large datasets
- **Alternative**: HDBSCAN for varying densities
  Key Insight: DBSCAN doesn't assume spherical clusters like K-means!

Other Clustering Algorithms

1. Agglomerative Clustering
   - **How**: Bottom-up merging of closest clusters
   - **Pros**:
     - Captures various shapes
     - Creates cluster hierarchy tree
     - Works with any distance metric
   - **Cons**:
     - Needs connectivity matrix for large datasets
     - Slow without connectivity matrix
2. BIRCH
   - **How**: Builds memory-efficient tree structure
   - **Pros**:
     - Fast for large datasets
     - Low memory usage
   - **Cons**:
     - Limited to <20 features
     - Similar results to k-means
3. Mean-Shift
   - **How**: Circles shift toward higher density until convergence
   - **Pros**:
     - Finds any number/shape of clusters
     - Only 1 hyperparameter (bandwidth)
   - **Cons**:
     - $O(m^2)$ complexity - slow for large datasets
     - Chops clusters with internal density variations
4. Affinity Propagation
   - **How**: Instances "vote" for exemplars to form clusters
   - **Pros**:
     - No need to specify cluster count

    - Handles different cluster sizes
- **Cons**:
    - $O(m^2)$ complexity - very slow for large datasets

5. Spectral Clustering
    - **How**: Reduce similarity matrix dimensions → apply k-means
    - **Pros**:
        - Captures complex structures
        - Good for graph clustering
    - **Cons**:
        - Doesn't scale well
        - Struggles with different cluster sizes

- Quick Comparison Table

Algorithm	Scalability	Cluster Shapes	Need k?	Best For
Agglomerative	Medium-	Any	No	Hierarchical data
BIRCH	High	Spherical	Yes	Large datasets
Mean-Shift	Low	Any	No	Density-based
Affinity Prop	Low	Any	No	Small datasets
Spectral	Low	Complex	Yes	Graph data

-With connectivity matrix

Gaussian Mixtures

1. Core Concept
    - **Definition:** A probabilistic model that assumes data instances are generated from a mixture of several Gaussian distributions with unknown parameters.
    - **Cluster Shape:** Unlike K-Means (which forces circles), GMM clusters can be **ellipsoids** of any shape, size, density, or orientation.
    - **Parameters to Learn:**
    - μ (Means): Center of the cluster.
    - Σ (Covariance Matrices): Shape, size, and orientation.
    - ϕ (Weights): Relative importance (density) of the cluster.

2. The Algorithm: Expectation-Maximization (EM) GMM uses the EM algorithm, which is a generalization of K-Means using **Soft Assignments**. It repeats two steps until convergence:
   - **Step 1: Expectation (E-Step):**
     - Calculate the probability (responsibility) that each instance belongs to each cluster based on current parameters.
   - **Step 2: Maximization (M-Step):**
     - Update the parameters (μ, Σ, ϕ) using all instances, weighted by the probabilities found in the E-step.
3. Code Implementation (Scikit-Learn)
   - **Initialization & Training**
     - `n_init` **is critical:** EM can get stuck in local optima (poor solutions). Setting `n_init=10` runs the algorithm 10 times and keeps the best one. Default is 1.

```
from sklearn.mixture import GaussianMixture

# n_components = k (must be known in advance)
# n_init = 10 (run 10 times to find best weights)
gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

   - **Checking Convergence**

```
# Did the model converge? (True/False)
gm.converged_

# How many iterations did it take?
gm.n_iter_
```

   - **Inspecting Learned Parameters**

```
gm.weights_      # The relative weight of each cluster
gm.means_        # The center points of clusters
gm.covariances_  # The matrices defining shape/orientation
```

4. Making Predictions
   - **Option A: Hard Clustering** Assigns the instance to the single cluster with the highest probability.

```
# Returns array of cluster indices [0, 0, 1, ...]
gm.predict(X)
```

   - **Option B: Soft Clustering** Returns the estimated probability vector for all clusters.

```
# Returns array of shape (n_samples, n_clusters)
# e.g., [[0.97, 0.02, 0.01], ...]
gm.predict_proba(X)
```

5. Generative & Density Capabilities
    - **Generative Model (Creating New Da3ta)** Since GMM learns the distribution, it can generate entirely new instances similar to the training data.

```
# Generate 6 new instances
# Returns X_new (features) and y_new (cluster labels)
X_new, y_new = gm.sample(6)
```

    - **Density Estimation (Anomaly Detection)** Estimates the log of the Probability Density Function (PDF).
        - **Output:** Usually negative numbers (log probabilities).
        - **Usage:** Lower scores = Lower density = Potential **Anomaly/Outlier**.

```
# Returns log-density for each instance
gm.score_samples(X).round(2)
```

- **Covariance Types** in GMMs
    - GMM Hyperparameter: `covariance_type`
    1. **Why use it?** When you have many dimensions (features) or few data points, the EM algorithm struggles to converge. By limiting the shape/freedom of the clusters, you reduce the number of parameters the model has to learn, making it more stable.
    2. **The 4 Types of Constraints** You set this using `covariance_type` in Scikit-Learn.
    -

Type	Shape Constraint	Orientation	Shared?	Complexity
`"full"` (Default)	Any Ellipsoid	Any Angle	No (Each unique)	**High**
`"tied"`	Any Ellipsoid	Any Angle	**Yes** (All clusters same shape)	**High**
`"spherical"`	Spheres	N/A	No (Diff sizes allowed)	**Low**
`"diag"`	Ellipsoids	**Axis-Aligned** (No rotation)	No	**Low**

    - **Visual Rule of Thumb:**
        - **Spherical:** Like K-Means (circles), but sizes can differ.
        - **Diag:** Stretched circles (ovals), but cannot be tilted diagonally.

- **Tied:** All clusters look like "clones" (same rotation/shape), just at different locations.
- **Full:** Total freedom.

3. **Computational Complexity (Big-O)** This helps you decide which to use based on your dataset size (m instances, n features, k clusters).
   - **Fast (O(kmn)):**
     - `"spherical"` and `"diag"`
     - Scales linearly with features (n). Good for high-dimensional data.
   - **Slow (O(kmn+kn3)):**
     - `"full"` and `"tied"`
     - Scales cubically (n3) with features. **Do not use** if you have thousands of features (e.g., raw pixels).
4. **Code Implementation**

```
from sklearn.mixture import GaussianMixture

# Example: Restricting clusters to be spherical (faster, less
overfitting)
gm = GaussianMixture(n_components=3, n_init=10,
covariance_type="spherical")
gm.fit(X)
```

Using Gaussian Mixtures for Anomaly Detection

1. **The Core Concept**
   - **Logic:** GMMs learn what "normal" data looks like (high density). Any data point located in a **low-density region** is considered an anomaly.
   - **Thresholding:** You must define a "density threshold" to decide what counts as an anomaly.
     - *Example:* If you know your manufacturing process produces 4% defective products, you set the threshold at the 4th percentile.
2. **Code Implementation** How to detect and isolate outliers:

```
# 1. Get the log probability density for every instance
densities = gm.score_samples(X)

# 2. Define the threshold (e.g., bottom 4% are anomalies)
# density_threshold is the specific log-value separating the bottom
4%
density_threshold = np.percentile(densities, 4)
```

```
# 3. Filter the dataset to find the anomalies
anomalies = X[densities < density_threshold]
```

3. **Tuning the Threshold (Precision/Recall Trade-off)**
    - **False Positives (Flagging good items as bad):** If you catch too many normal items, **lower** the threshold.
    - **False Negatives (Missing bad items):** If you miss actual defects, **increase** the threshold.
4. **Important Distinction: Anomaly vs. Novelty Detection**
    - **Anomaly Detection:** Assumes the training dataset **contains outliers** (is mixed). The goal is to clean them up or find them.
    - **Novelty Detection:** Assumes the training dataset is **100% clean**. The goal is to detect *new* types of data (novelties) that differ from the training set.
5. **Pro Tip: Handling Bias**
    - **The Problem:** GMM tries to fit *all* data points, including outliers. If there are too many outliers, the model stretches its shape to include them, making them look "normal."
    - **The Fix (Iterative Approach):**
        1. Fit the model once.
        2. Detect and remove the worst outliers.
        3. **Refit** the model on the cleaned dataset.
    - **Alternative:** Use `sklearn.covariance.EllipticEnvelope` for robust covariance estimation.

Selecting the Number of Clusters

1. Why not use K-Means methods?
    - **Inertia & Silhouette Score:** These metrics rely on clusters being spherical and roughly the same size.
    - **The Problem:** Since GMMs form **ellipsoids** of varying sizes, these standard metrics are unreliable and often wrong.
2. The Solution: Information Criteria (AIC & BIC) Instead of measuring distance, we measure the balance between **Model Fit** and **Model Complexity**.
    - **Goal:** Find the k that **MINIMIZES** the AIC or BIC score.
    - **Rule:** Lower score = Better model.
3. The Variables
    - m: Number of instances (data points).
    - p: Number of parameters (complexity).

    - *Note:* More clusters = High p (bad for score).
  - $\hat{l}$: Maximized Likelihood (how well it fits).
    - *Note:* Better fit = High $\hat{l}$ (good for score).
4. The Formulas Both formulas penalize high complexity (p) and reward high fit ($\hat{l}$).
   1. BIC (Bayesian Information Criterion)
      - **The Formula:**

$$BIC = Log(m) * p - 2 * log(\hat{L})$$

      - **The Personality:** Strict & Conservative.
      - **The Behavior:**
        - It applies a **heavy penalty** on complexity (the number of parameters p).
        - The penalty grows significantly as your dataset size (m) grows.
      - **The Outcome:** It prefers **simpler models** (fewer clusters). It would rather underfit slightly than overfit.
   2. AIC (Akaike Information Criterion)
      - **The Formula:**

$$AIC = 2 * p - 2 * log(\hat{L})$$

      - **The Personality:** Lenient & Flexible.
      - **The Behavior:**
        - It applies a **light penalty** on complexity (only 2×p).
        - It focuses more on how well the model fits the data (L^) than on keeping it simple.
      - **The Outcome:** It may suggest **complex models** (more clusters). It tolerates a bit of overfitting if it means getting a better fit.
5. How to select the best k?
   1. Train GMMs with different numbers of clusters (e.g., k=1 to 10).
   2. Compute AIC and BIC for each.
   3. Plot the scores.
   4. Choose the k where the score is **lowest** (the minimum point).

- Code Implementation

```
import matplotlib.pyplot as plt

# 1. Train models with range of clusters (e.g., 1 to 10)
```

```
n_components = np.arange(1, 11)
models = [GaussianMixture(n, n_init=10, random_state=42).fit(X)
          for n in n_components]

# 2. Calculate AIC and BIC for each model
bic_scores = [m.bic(X) for m in models]
aic_scores = [m.aic(X) for m in models]

# 3. Plot the results
plt.figure(figsize=(8, 4))
plt.plot(n_components, bic_scores, label='BIC', color='blue')
plt.plot(n_components, aic_scores, label='AIC', color='green',
linestyle='--')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Information Criterion Score')
plt.legend()
plt.title('Minimize AIC/BIC to find optimal k')
plt.show()

# The optimal k is the lowest point on these lines.
```

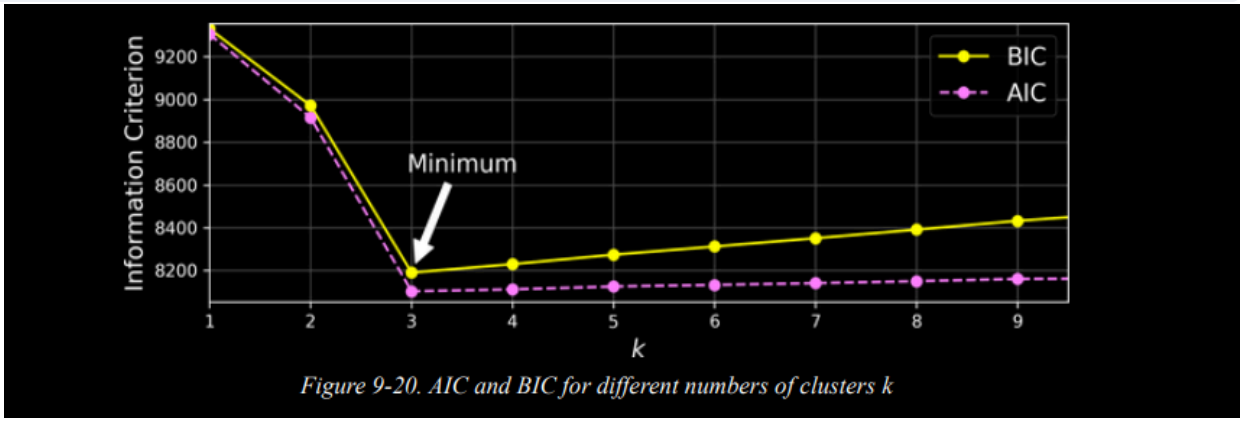
- 

Figure 9-20. AIC and BIC for different numbers of clusters k

Bayesian Gaussian Mixture Models

The Problem: Standard GMM requires you to guess the number of clusters (k) or run multiple tests with AIC/BIC. **The Solution:** `BayesianGaussianMixture` can automatically detect the correct number of clusters.

- **How it works:**
  1. You initialize the model with a **larger than necessary** number of clusters (e.g., k=10).
  2. The algorithm learns which clusters are actually needed.
  3. It assigns a **weight of 0** to the unnecessary clusters, effectively ignoring them.
- **Code Implementation:**

```python
from sklearn.mixture import BayesianGaussianMixture

# 1. Set n_components to a number higher than you expect (e.g., 10)
bgm = BayesianGaussianMixture(n_components=10, n_init=10,
random_state=42)
bgm.fit(X)

# 2. Check the weights.
# Notice how many become 0. Those are the discarded clusters.
print(bgm.weights_.round(2))
# Output: [0.4, 0.21, 0.4, 0. , 0. , 0. , ... ]
# Result: Only 3 active clusters found.
```

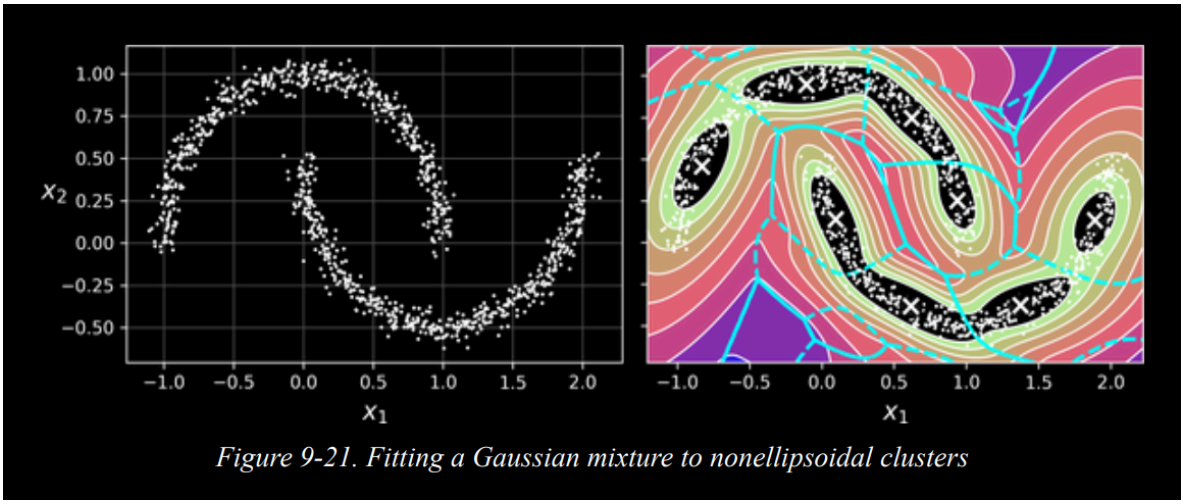
- Critical Limitation: The "Ellipsoid" Constraint
  The Issue: GMMs assume that all clusters are **Ellipsoids** (Gaussian/Normal distributions). They cannot adapt to irregular shapes.
  - **The "Moons" Example (Figure 9-21):**
    - 

      Figure 9-21. Fitting a Gaussian mixture to nonellipsoidal clusters
    - **Input:** Two interlocking crescent moons (non-ellipsoidal).
    - **Expectation:** 2 Clusters (Top moon, Bottom moon).
    - **Reality:** The model found **8 separate clusters**.
  - **Why did it fail?** Since a single Gaussian cannot bend into a "C" shape, the model was forced to stack many small ovals together to cover the data points.
  - **Takeaway:**
    - **Clustering: FAILED.** It cannot recognize that the top moon is one single group.
    - **Anomaly Detection: PASSED.** The model still learned the *density* of the data (the colored heatmap fits the shape well), so it can still detect if a new point is an outlier.

Resources :

-

Related notes :

-

References :

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