# Unsupervised Learning - Coding Practice Questions

CM52054: Foundational Machine Learning
*Practice set with fully worked answers*

1) **Implement k-means from scratch.**
Implement k-means clustering with:

- centroid initialisation by sampling $K$ *distinct data points*,
- Euclidean distance,
- stopping when assignments stop changing *or* centroid movement is below a tolerance.

Return `(centroids, labels, wcss)`, where

$$\text{WCSS (Within-Cluster Sum of Squares)} = \sum_{i=1}^{N} \|x_i - c_{\ell_i}\|_2^2.$$

**Answer:**

```
import numpy as np

def kmeans_from_scratch(X, K, max_iter=100, tol=1e-6, seed=0):
    """
    Simple, easy-to-follow k-means (NumPy only).

    Returns
    -------
    centroids : (K, M) array
    labels : (N,) array of ints in {0,...,K-1}
    wcss : float, sum of squared distances to assigned centroid
    """
    # ---------- Step 0: basic setup ----------
    X = np.asarray(X, dtype=float)
    N, M = X.shape
    rng = np.random.default_rng(seed)

    # ---------- Step 1: initialise centroids ----------
    # Pick K random data points as initial centroids
    centroids = X[rng.choice(N, size=K, replace=False)].copy()

    # ---------- Step 2: repeat assignment + update ----------
    for _ in range(max_iter):
        # (A) Assignment: compute distance to each centroid, pick nearest
        # dists_sq[i, k] = ||X[i] - centroids[k]||^2
        dists_sq = np.zeros((N, K))
        for k in range(K):
            diff = X - centroids[k] # (N, M)
            dists_sq[:, k] = np.sum(diff**2, axis=1)

        labels = np.argmin(dists_sq, axis=1) # (N,)
```

```python

        # (B) Update: recompute each centroid as mean of its assigned
            points
        new_centroids = centroids.copy()
        for k in range(K):
            points_k = X[labels == k]
            if len(points_k) > 0:
                new_centroids[k] = points_k.mean(axis=0)
            else:
                # If a cluster becomes empty, re-pick a random data
                    point
                new_centroids[k] = X[rng.integers(0, N)]

        # (C) Convergence: stop if centroids barely move
        shift = np.linalg.norm(new_centroids - centroids)
        centroids = new_centroids
        if shift < tol:
            break

    # ---------- Step 3: compute WCSS ----------
    # Sum of squared distances of each point to its assigned
        centroid
    wcss = 0.0
    for i in range(N):
        diff = X[i] - centroids[labels[i]]
        wcss += float(np.sum(diff**2))

    return centroids, labels, wcss


if __name__ == "__main__":
    X = np.array([[0.0, 0.0],
                  [0.2, 0.1],
                  [3.0, 3.0],
                  [3.1, 2.9],
                  [10.0, 10.0]])
    centroids, labels, wcss = kmeans_from_scratch(X, K=2, seed=42)
    print("Centroids:\n", centroids)
    print("Labels:", labels)
    print("WCSS:", wcss)
```

2) **Best-of-$n$ restarts for k-means (mitigate non-determinism).**
Implement a wrapper `kmeans_best_of_n` that:

- runs k-means `n_init` times with different seeds,
- returns the solution with the *lowest* WCSS.

**Answer:**

```python
import numpy as np

def kmeans_best_of_n(X, K, n_init=10, max_iter=100, tol=1e-6,
    base_seed=0):
    """
    Run k-means multiple times and choose the result with the lowest
        WCSS.

```

```python
    Parameters
    ----------
    X : array-like, shape (N, M)
    K : int
    n_init : int
        Number of independent initialisations.
    max_iter, tol : forwarded to kmeans_from_scratch
    base_seed : int
        Base seed; each run uses base_seed + i.

    Returns
    -------
    best_centroids : np.ndarray, shape (K, M)
    best_labels : np.ndarray, shape (N,)
    best_wcss : float
    """
    X = np.asarray(X, dtype=float)

    best_centroids = None
    best_labels = None
    best_wcss = np.inf

    # Run k-means with different seeds, inducing different initial centroid choices.
    for i in range(n_init):
        seed = base_seed + i

        # Call the k-means implementation from Q1.
        centroids, labels, wcss = kmeans_from_scratch(
            X, K, max_iter=max_iter, tol=tol, seed=seed
        )

        # Keep the solution with the smallest objective value.
        if wcss < best_wcss:
            best_wcss = wcss
            best_centroids = centroids
            best_labels = labels

    return best_centroids, best_labels, float(best_wcss)


if __name__ == "__main__":
    rng = np.random.default_rng(0)

    # Create two clusters for demonstration
    X1 = rng.normal(loc=(0, 0), scale=0.5, size=(50, 2))
    X2 = rng.normal(loc=(5, 5), scale=0.5, size=(50, 2))
    X = np.vstack([X1, X2])

    centroids, labels, wcss = kmeans_best_of_n(X, K=2, n_init=20, base_seed=100)
    print("Best WCSS:", wcss)
```

3) **Compute "good clustering" metrics (intra vs inter).**

Implement a function `clustering_quality_metrics(X, labels, centroids)` that computes:

(a) average intra-cluster distance: mean $\|x_i - c_{\ell_i}\|_2$,

(b) minimum inter-centroid distance: $\min_{k \neq k'} \|c_k - c_{k'}\|_2$.

**Answer:**

```python
import numpy as np

def clustering_quality_metrics(X, labels, centroids):
    """
    Simple, easy-to-follow clustering metrics:

    1) avg_intra: average distance from each point to its assigned centroid
    2) min_inter: minimum distance between any two centroids
    """
    X = np.asarray(X, dtype=float)
    labels = np.asarray(labels, dtype=int)
    centroids = np.asarray(centroids, dtype=float)

    N = X.shape[0]
    K = centroids.shape[0]

    # -----------------------------
    # 1) Average intra-cluster distance
    # -----------------------------
    # For each point i, compute distance to its assigned centroid.
    total = 0.0
    for i in range(N):
        c = centroids[labels[i]] # centroid of point i
        diff = X[i] - c
        dist = np.sqrt(np.sum(diff**2)) # Euclidean distance
        total += float(dist)
    avg_intra = total / N

    # -----------------------------
    # 2) Minimum inter-centroid distance
    # -----------------------------
    # Compute distance between every pair of centroids and take the smallest.
    min_inter = float("inf")
    for i in range(K):
        for j in range(i + 1, K):
            diff = centroids[i] - centroids[j]
            dist = np.sqrt(np.sum(diff**2))
            if dist < min_inter:
                min_inter = float(dist)

    return avg_intra, min_inter


if __name__ == "__main__":
    rng = np.random.default_rng(0)
    X = rng.normal(size=(20, 2))

    # Example: two fixed centroids and nearest-centroid labels
    centroids = np.array([[0.0, 0.0], [1.0, 1.0]])
    labels = np.zeros(X.shape[0], dtype=int)
    for i in range(X.shape[0]):
```

```python
        d0 = np.sum((X[i] - centroids[0])**2)
        d1 = np.sum((X[i] - centroids[1])**2)
        labels[i] = 0 if d0 <= d1 else 1

    avg_intra, min_inter = clustering_quality_metrics(X, labels, centroids)
    print("Average intra distance:", avg_intra)
    print("Minimum inter-centroid distance:", min_inter)
```

4) **Optional: Agglomerative hierarchical clustering (single linkage) for small datasets.** Implement agglomerative clustering with single linkage:

- start with each point as its own cluster,
- repeatedly merge the pair of clusters with minimum single-link distance,
- return a merge history list.

Assume $N$ is small; clarity is more important than speed.

**Answer:**

```python
import numpy as np

def euclidean(a, b):
    """Euclidean distance between two vectors a and b."""
    return float(np.sqrt(np.sum((a - b) ** 2)))

def single_link_distance(X, cluster_a, cluster_b):
    """
    Single-link distance between two clusters:
    the minimum distance between any point in cluster_a and any point in cluster_b.
    """
    best = float("inf")
    for i in cluster_a:
        for j in cluster_b:
            d = euclidean(X[i], X[j])
            if d < best:
                best = d
    return best

def agglomerative_single_linkage(X):
    """
    Easy-to-follow agglomerative clustering (single linkage).

    Returns
    -------
    merges : list of (a_id, b_id, dist, new_id)
        a_id and b_id are the cluster IDs merged at distance dist,
        producing a new cluster with ID new_id.
    """
    X = np.asarray(X, dtype=float)
    N = X.shape[0]

    # Start with N singleton clusters: {0}, {1}, ..., {N-1}
    clusters = {i: {i} for i in range(N)} # cluster_id -> set of point indices
    next_id = N
    merges = []
```

```
    # Keep merging until only one cluster remains
    while len(clusters) > 1:
        ids = list(clusters.keys())

        # Find the closest pair of clusters
        best_dist = float("inf")
        best_pair = None

        for i in range(len(ids)):
            for j in range(i + 1, len(ids)):
                a_id = ids[i]
                b_id = ids[j]
                d = single_link_distance(X, clusters[a_id], clusters[b_id])
                if d < best_dist:
                    best_dist = d
                    best_pair = (a_id, b_id)

        # Merge the closest pair
        a_id, b_id = best_pair
        new_cluster = clusters[a_id] | clusters[b_id]

        new_id = next_id
        next_id += 1

        # Record this merge event
        merges.append((a_id, b_id, best_dist, new_id))

        # Update active clusters
        del clusters[a_id]
        del clusters[b_id]
        clusters[new_id] = new_cluster

    return merges


if __name__ == "__main__":
    X = np.array([
        [0.0, 0.0],
        [0.1, 0.0],
        [2.0, 2.0],
        [2.1, 2.0],
    ])

    merges = agglomerative_single_linkage(X)
    for a_id, b_id, dist, new_id in merges:
        print(f"Merged {a_id} and {b_id} at dist={dist:.3f} -> new cluster {new_id}")
```