CWID: A20555359

CS422 Data Mining
Assignment 3

1. Recitation Exercises

1.1 Chapter 13

1) **Given the data points:**
2, 4, 10, 12, 3, 20, 30, 11, 25

Run one iteration of K-means with:
k = 3

Initial means (centroids):
$\mu 1 = 2$, $\mu 2 = 4$, $\mu 3 = 6$

Step 1: Assign each point x_i to the closest centroid μ_j
Calculate distance $d(x_i, \mu_j) = |x_i - \mu_j|$ for all j = 1,2,3

Point x = 2:
$d(2, \mu 1) = |2 - 2| = 0$
$d(2, \mu 2) = |2 - 4| = 2$
$d(2, \mu 3) = |2 - 6| = 4$
Closest centroid: $\mu 1 \rightarrow$ Cluster 1

Point x = 4:
$d(4, \mu 1) = |4 - 2| = 2$
$d(4, \mu 2) = |4 - 4| = 0$
$d(4, \mu 3) = |4 - 6| = 2$
Closest centroid: $\mu 2 \rightarrow$ Cluster 2

Point x = 10:
$d(10, \mu 1) = |10 - 2| = 8$
$d(10, \mu 2) = |10 - 4| = 6$
$d(10, \mu 3) = |10 - 6| = 4$
Closest centroid: $\mu 3 \rightarrow$ Cluster 3

Point x = 12:
$d(12, \mu 1) = |12 - 2| = 10$

d(12, μ2) = |12 - 4| = 8
d(12, μ3) = |12 - 6| = 6
Closest centroid: μ3 → Cluster 3

Point x = 3:
d(3, μ1) = |3 - 2| = 1
d(3, μ2) = |3 - 4| = 1
d(3, μ3) = |3 - 6| = 3
Tie between μ1 and μ2 (both distance = 1)
Break tie arbitrarily → assign to μ1 → Cluster 1

Point x = 20:
d(20, μ1) = |20 - 2| = 18
d(20, μ2) = |20 - 4| = 16
d(20, μ3) = |20 - 6| = 14
Closest centroid: μ□ → Cluster 3

Point x = 30:
d(30, μ1) = |30 - 2| = 28
d(30, μ2) = |30 - 4| = 26
d(30, μ3) = |30 - 6| = 24
Closest centroid: μ3 → Cluster 3

Point x = 11:
d(11, μ1) = |11 - 2| = 9
d(11, μ2) = |11 - 4| = 7
d(11, μ3) = |11 - 6| = 5
Closest centroid: μ3 → Cluster 3

Point x = 25:
d(25, μ1) = |25 - 2| = 23
d(25, μ2) = |25 - 4| = 21
d(25, μ3) = |25 - 6| = 19
Closest centroid: μ3→ Cluster 3

Step 2: Form clusters with assigned points

Cluster 1 (μ1): {2, 3}
Cluster 2 (μ2): {4}
Cluster 3 (μ3): {10, 12, 20, 30, 11, 25}

Step 3: Compute new means for each cluster:

New μ1 = (2 + 3) / 2 = 5 / 2 = 2.5
New μ2 = 4 / 1 = 4.0
New μ3= (10 + 12 + 20 + 30 + 11 + 25) / 6 = 108 / 6 = 18.0

Final Result after 1 iteration:

Cluster 1: {2, 3} $\rightarrow$ $\mu1 = 2.5$
Cluster 2: {4} $\rightarrow$ $\mu2 = 4.0$
Cluster 3: {10, 12, 20, 30, 11, 25} $\rightarrow$ $\mu3 = 18.0$

2) Given Dataset:

$x1^T = [0, 2]$
$x2^T = [0, 0]$
$x3^T = [1.5, 0]$
$x4^T = [5, 0]$
$x5^T = [5, 2]$ d_1

Initial clusters:
C1= { x1, x2, x4}
C2= { x3, x5 }

k = 2

Part (a-1): K-Means with Euclidean distance (L2 norm)

Step 1: Compute initial centroids

$\mu1 = (1/|C1|)\cdot(x1 + x2 + x4)$
$= (1/3)\cdot([0,2] + [0,0] + [5,0])$
$= [\ (0+0+5)/3,\ (2+0+0)/3\ ]$
$\mathbf{= [5/3,\ 2/3]}$

$\mu2= (1/|C2|)\cdot(x3+ x5)$
$= (1/2)\cdot([1.5,0] + [5,2])$
$= [\ (1.5+5)/2,\ (0+2)/2\ ]$
$= [6.5/2,\ 2/2]$
$\mathbf{= [3.25,\ 1]}$

Step 2: Assign each x_i to nearest centroid using

$d2(x,\mu) = \sqrt{[\ (x1-\mu1)^2 + (x2-\mu2\ )^2\ ]}$

For x1 = [0,2]:

$d2(x1,\mu1) = \sqrt{[\ (0-5/3)^2 + (2-2/3)^2\ ]}$
$= \sqrt{[\ (-5/3)^2 + (4/3)^2\ ]}$
$= \sqrt{[25/9 + 16/9]}$
$= \sqrt{[41/9]}$
$= \sqrt{41}/3$

$d2(x1,\mu2) = \sqrt{[\ (0-3.25)^2 + (2-1)^2\ ]}$
$= \sqrt{[\ (-3.25)^2 + 1^2\ ]}$
$= \sqrt{[10.5625 + 1\ ]}$

$= \sqrt{11.5625}$

$\rightarrow$ assign x1 to C1 ($\sqrt{41}/3 \approx 2.14 < \sqrt{11.56} \approx 3.40$)

Repeat for x2, x3, x4, x5:

x2 = [0,0]:

$d2(\cdot,\mu 1) = \sqrt{[(-5/3)^2 + (-2/3)^2]} = \sqrt{[25/9 + 4/9]} = \sqrt{[29/9]}$

$d2(\cdot,\mu 2) = \sqrt{[10.5625 + 1]} = \sqrt{11.5625}$

$\rightarrow$ **C1**

x3= [1.5,0]:

$d2(\cdot,\mu 1) = \sqrt{[(1.5-5/3)^2 + (-2/3)^2]}$

$= \sqrt{[(-1/6)^2 + (-2/3)^2]}$

$= \sqrt{[1/36 + 4/9]} = \sqrt{[1/36 + 16/36]} = \sqrt{[17/36]} = \sqrt{17}/6$

$d2(\cdot,\mu 2) = \sqrt{[(1.5-3.25)^2 + (-1)^2]} = \sqrt{[(-1.75)^2 + 1]} = \sqrt{[3.0625 + 1]}$

$\rightarrow$ **C1**

x4= [5,0]:

$d2(\cdot,\mu 1) = \sqrt{[(5-5/3)^2 + (-2/3)^2]} = \sqrt{[(10/3)^2 + 4/9]}$

$d2(\cdot,\mu 2) = \sqrt{[(5-3.25)^2 + (-1)^2]} = \sqrt{[1.75^2 + 1]}$

$\rightarrow$ **C2**

x5 = [5,2]:

$d2(\cdot,\mu 1) = \sqrt{[(10/3)^2 + (4/3)^2]}$

$d2(\cdot,\mu 2) = \sqrt{[1.75^2 + 1]}$

$\rightarrow$ **C2**

Step 3: Form new clusters

C1 = { x1, x2, x3}

C2 = { x4, x5 }

Step 4: Recompute centroids

μ1 = (1/3)·([0,2] + [0,0] + [1.5,0])

= [(0+0+1.5)/3, (2+0+0)/3]

= **[0.5, 2/3]**

μ2 = (1/2)·([5,0] + [5,2])

= **[5, 1]**

Step 5: Reassign and check

Distances to new μ's give same assignments $\rightarrow$ converged

Final (Euclidean):

C1 = {x1, x2, x3}, μ1 = [0.5, 0.67]

C2 = {x4, x5}, μ2 = [5, 1]

Part (a-2): K-Means with Manhattan distance (L_1 norm)

Step 1: Use same initial centroids as in (a-1)

$\mu1 = [5/3, 2/3]$
$\mu2 = [3.25, 1]$

Step 2: Assign each x_i using Manhattan distance

$d_1(x, \mu) = |x1–\mu1| + |x2–\mu2|$

x1 = [0,2]:

$d1(x1, \mu1) = |0–5/3| + |2–2/3| = 5/3 + 4/3 = 9/3 = 3$
$d1(x1, \mu2) = |0–3.25| + |2–1| = 3.25 + 1 = 4.25$
$\rightarrow$ assign x1 $\rightarrow$ C1

x2 = [0,0]:

$d1(x2, \mu1) = |0–5/3| + |0–2/3| = 5/3 + 2/3 = 7/3 \approx 2.33$
$d1(x2, \mu2) = |0–3.25| + |0–1| = 3.25 + 1 = 4.25$
$\rightarrow$ assign x2 $\rightarrow$ C1

x3= [1.5, 0]:

$d1(x3, \mu1) = |1.5–5/3| + |0–2/3| = |–1/6| + 2/3 = 1/6 + 4/6 = 5/6 \approx 0.83$
$d1(x3, \mu2) = |1.5–3.25| + |0–1| = 1.75 + 1 = 2.75$
$\rightarrow$ assign x3$\rightarrow$ C1

x4= [5,0]:

$d1(x4, \mu1) = |5–5/3| + |0–2/3| = 10/3 + 2/3 = 12/3 = 4$
$d1(x4, \mu2) = |5–3.25| + |0–1| = 1.75 + 1 = 2.75$
$\rightarrow$ assign x4$\rightarrow$ C2

x5 = [5,2]:

$d1(x5, \mu1) = |5–5/3| + |2–2/3| = 10/3 + 4/3 = 14/3 \approx 4.67$
$d1(x5, \mu2) = |5–3.25| + |2–1| = 1.75 + 1 = 2.75$
$\rightarrow$ assign x5 $\rightarrow$ C2

Step 3: New clusters after assignment

C1 = {x1, x2, x3}
C2 = {x4, x5}

Step 4: Recompute centroids

$\mu1 = (1/3)\cdot([0,2] + [0,0] + [1.5,0]) = [0.5, 2/3]$
$\mu2 = (1/2)\cdot([5,0] + [5,2]) = [5, 1]$

Step 5: Reassign and check

$\rightarrow$ distances yield same cluster assignments $\rightarrow$ converged

Final Output (Manhattan):
C1 = {x1, x2, x3}, $\mu 1 = [0.5, 0.67]$
C2 = {x4, x5}, $\mu 2 = [5, 1]$

Part (b): EM Algorithm — One iteration

Assume:
$\mu 1 = [0.5, 0.67]$
$\mu 2 = [5.0, 1.0]$
$\pi 1 = \pi 2 = 0.5$
$\sigma^2 = 1$ (same for all dimensions)

Multivariate normal density (with diagonal covariance):
$p(x \mid \mu) = (1 / (2\pi)) \cdot \exp(-0.5 \cdot \|x - \mu\|^2)$

We compute numerator and denominator of:
$\gamma_j 1 = (\pi 1 \cdot p(x_j \mid \mu 1)) / (\pi 1 \cdot p(x_j \mid \mu 1) + \pi 2 \cdot p(x_j \mid \mu 2))$
$\gamma_j 2 = 1 - \gamma_j 1$

Compute each point:
x1 = [0, 2]
$\|x1 - \mu 1\|^2 = (0 - 0.5)^2 + (2 - 0.67)^2 = 0.25 + 1.7556 = 2.0056$
$\|x1 - \mu 2\|^2 = (0 - 5)^2 + (2 - 1)^2 = 25 + 1 = 26$

$p(x1 \mid \mu 1) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 2.0056) = (1 / 2\pi) \cdot \exp(-1.0028) \approx 0.0585$
$p(x1 \mid \mu 2) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 26) = (1 / 2\pi) \cdot \exp(-13) \approx 2.26 \times 10^{-7}$

$\gamma 11 = (0.5 \cdot 0.0585) / (0.5 \cdot 0.0585 + 0.5 \cdot 2.26 \times 10^{-7})$
$= 0.02925 / (0.02925 + 1.13 \times 10^{-7}) \approx 0.999996$

$\gamma 1 \approx [1.0000, 0.0000]$

x2 = [0, 0]
$\|x2 - \mu 1\|^2 = (0 - 0.5)^2 + (0 - 0.67)^2 = 0.25 + 0.4489 = 0.6989$
$\|x2 - \mu 2\|^2 = (0 - 5)^2 + (0 - 1)^2 = 25 + 1 = 26$

$p(x2 \mid \mu 1) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 0.6989) \approx 0.0829$
$p(x2 \mid \mu 2) =$ same as before $\approx 2.26 \times 10^{-7}$

$\gamma 21 = (0.5 \cdot 0.0829) / (0.5 \cdot 0.0829 + 0.5 \cdot 2.26 \times 10^{-7})$
$\approx 0.04145 / (0.04145 + 1.13 \times 10^{-7}) \approx 0.999997$

$\gamma_2 \approx [1.0000, 0.0000]$

$x_3 = [1.5, 0]$

$\|x_3 - \mu_1\|^2 = (1.5 - 0.5)^2 + (0 - 0.67)^2 = 1.0 + 0.4489 = 1.4489$

$\|x_3 - \mu_2\|^2 = (1.5 - 5)^2 + (0 - 1)^2 = 12.25 + 1 = 13.25$

$p(x_3 | \mu_1) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 1.4489) \approx 0.0654$

$p(x_3 | \mu_2) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 13.25) \approx 1.18 \times 10^{-3}$

$\gamma_{31} = (0.5 \cdot 0.0654) / (0.5 \cdot 0.0654 + 0.5 \cdot 1.18 \times 10^{-3})$

$= 0.0327 / (0.0327 + 0.00059) \approx 0.9973$

$\gamma_3 \approx [0.9973, 0.0027]$

$x_4 = [5, 0]$

$\|x_4 - \mu_1\|^2 = (5 - 0.5)^2 + (0 - 0.67)^2 = 20.25 + 0.4489 = 20.6989$

$\|x_4 - \mu_2\|^2 = (5 - 5)^2 + (0 - 1)^2 = 0 + 1 = 1$

$p(x_4 | \mu_1) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 20.6989) \approx 4.89 \times 10^{-4}$

$p(x_4 | \mu_2) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 1) \approx 0.0965$

$\gamma_{41} = (0.5 \cdot 4.89 \times 10^{-4}) / (0.5 \cdot 4.89 \times 10^{-4} + 0.5 \cdot 0.0965)$

$= 2.445 \times 10^{-4} / (2.445 \times 10^{-4} + 0.04825) \approx 0.000005$

$\gamma_4 \approx [0.0001, 0.9999]$

$x_5 = [5, 2]$

$\|x_5 - \mu_1\|^2 = (5 - 0.5)^2 + (2 - 0.67)^2 = 20.25 + 1.7556 = 22.0056$

$\|x_5 - \mu_2\|^2 = (5 - 5)^2 + (2 - 1)^2 = 0 + 1 = 1$

$p(x_5 | \mu_1) = (1 / 2\pi) \cdot \exp(-0.5 \cdot 22.0056) \approx 2.69 \times 10^{-5}$

$p(x_5 | \mu_2)$ = same as $x_4 \approx 0.0965$

$\gamma_{51} = (0.5 \cdot 2.69 \times 10^{-5}) / (0.5 \cdot 2.69 \times 10^{-5} + 0.5 \cdot 0.0965)$

$= 1.345 \times 10^{-5} / (1.345 \times 10^{-5} + 0.04825) \approx 0.00000027$

$\gamma_5 \approx [0.0000, 1.0000]$

Final Output:

$\gamma_1 \approx [1.0000, 0.0000]$

$\gamma_2 \approx [1.0000, 0.0000]$

$\gamma_3 \approx [0.9973, 0.0027]$

$\gamma_4 \approx [0.0001, 0.9999]$
$\gamma_5 \approx [0.0000, 1.0000]$

<u>M-Step: Update Cluster Means</u>

Let the data points be:
x1 = {0, 2}
x2 = {0, 0}
x3= {1.5, 0}
x4= {5, 0}
x5 = {5, 2}

Let the responsibilities (γ_{ji}) be:

γ_{11}= 1.0000	γ_{12} = 0.0000
γ_{21}= 1.0000	γ_{22}= 0.0000
γ_{31}= 0.9973	γ_{32}= 0.0027
γ_{41}= 0.0001	γ_{42}= 0.9999
γ_{51}= 0.0000	γ_{52}= 1.0000

Compute updated mean for cluster 1 (μ_1):

$\mu_1 = (\gamma_{11}\cdot x_1 + \gamma_{21}\cdot x_2 + \gamma_{31}\cdot x_3 + \gamma_{41}\cdot x_4 + \gamma_{51}\cdot x_5) / (\gamma_{11} + \gamma_{21} + \gamma_{31} + \gamma_{41} + \gamma_{51})$

$\mu_1 = (1\cdot\{0, 2\} + 1\cdot\{0, 0\} + 0.9973\cdot\{1.5, 0\} + 0.0001\cdot\{5, 0\} + 0\cdot\{5, 2\}) / (1 + 1 + 0.9973 + 0.0001 + 0)$

$\mu_1 = (\{0, 2\} + \{0, 0\} + \{1.49595, 0\} + \{0.0005, 0\} + \{0, 0\}) / 2.9974$

$\mu_1 = \{1.49645, 2\} / 2.9974$

$\mu_1 \approx \{0.4992, 0.6673\}$

Compute updated mean for cluster 2 (μ_2):

$\mu_2 = (\gamma_{12}\cdot x_1 + \gamma_{22}\cdot x_2 + \gamma_{32}\cdot x_3 + \gamma_{42}\cdot x_4 + \gamma_{52}\cdot x_5) / (\gamma_{12} + \gamma_{22} + \gamma_{32} + \gamma_{42} + \gamma_{52})$

$\mu_2 = (0\cdot\{0, 2\} + 0\cdot\{0, 0\} + 0.0027\cdot\{1.5, 0\} + 0.9999\cdot\{5, 0\} + 1\cdot\{5, 2\}) / (0 + 0 + 0.0027 + 0.9999 + 1)$

$\mu_2 = (\{0, 0\} + \{0, 0\} + \{0.00405, 0\} + \{4.9995, 0\} + \{5, 2\}) / 2.0026$

$\mu2 = \{10.00355, 2\} / 2.0026$

$\mu2 \approx \{4.9952, 0.9987\}$

Final Output:
$\mu1 = (\Sigma_j \gamma_j1 \cdot x_j) / (\Sigma_j \gamma_j1) \approx [0.4992, 0.6673]$
$\mu2 = (\Sigma_j \gamma_j2 \cdot x_j) / (\Sigma_j \gamma_j2) \approx [4.9952, 0.9987]$

1.2 Chapter 14

1) **Given,**
PointData = {
  x1 -> {1, 0, 1, 1, 0},
  x2 -> {1, 1, 0, 1, 0},
  x3 -> {0, 0, 1, 1, 0},
  x4 -> {0, 1, 0, 1, 0},
  x5 -> {1, 0, 1, 0, 1},
  x6 -> {0, 1, 1, 0, 0}
};

Step 1: Define similarity measures based on contingency table for two points x_i and x_j:

	x_j = 1	x_j = 0
x_i=1	n11	n10
x_i=0	n01	n00

- n11= number of attributes where both x_i and x_j are 1
- n10= number of attributes where x_i=1 and x_j=0
- n01= number of attributes where x_i=0 and x_j=1
- n00 = number of attributes where both x_i and x_j are 0

Similarity formulas:
- Simple Matching Coefficient (SMC):
$SMC(x_i, x_j) = (n11+ n00) / (n11+ n10+ n01+ n00)$

- Jaccard Coefficient (JC):
$JC(x_i, x_j) = n11/ (n11+ n10+ n01)$

- Rao's Coefficient (RC):
$RC(x_i, x_j) = n11/ (n11+ n10+ n01+ n00)$

Step 2: Calculate contingency values for each pair
Example: Calculate contingency values between x1 and x2

Attribute	**x1**	**x2**	**Match?**	**Category**
X1	**1**	**1**	**Both 1**	**n11 +1**
X2	**0**	**1**	**0 vs 1**	**n01+1**
X3	**1**	**0**	**1 vs 0**	**n10+1**
X4	**1**	**1**	**Both 1**	**n11+1**
X5	**0**	**0**	**Both 0**	**n00+1**

Therefore:
n11= 2, n10 = 1, n01 = 1, n00 = 1

Calculate similarities:
- SMC = (2 + 1) / 5 = 3/5 = 0.6
- JC = 2 / (2 + 1 + 1) = 2/4 = 0.5
- RC = 2 / 5 = 0.4

Total number of attributes = 5

Step 2: Contingency Table Values for All Pairs

Pair	n11	n10	n01	n00
(x1,x2)	2	1	1	1
(x1,x3)	2	1	0	2
(x1,x4)	1	2	1	1
(x1,x5)	3	0	1	1
(x1,x6)	1	2	2	0
(x2,x3)	1	2	1	1
(x2,x4)	2	1	1	1
(x2,x5)	1	2	0	2
(x2,x6)	2	1	2	0
(x3,x4)	1	1	1	2
(x3,x5)	2	1	1	1
(x3,x6)	2	0	1	2
(x4,x5)	1	2	1	1
(x4,x6)	2	0	1	2
(x5,x6)	1	1	2	1

Step 3: Similarity Values

Similarity Matrix (Rounded to 3 decimals)

Pair	SMC	JC	RC
(x1,x2)	0.6	0.5	0.4

(x1,x3)	0.8	0.667	0.4
(x1,x4)	0.4	0.25	0.2
(x1,x5)	0.8	0.75	0.6
(x1,x6)	0.2	0.25	0.2
(x2,x3)	0.4	0.25	0.2
(x2,x4)	0.6	0.5	0.4
(x2,x5)	0.6	0.333	0.2
(x2,x6)	0.4	0.4	0.4
(x3,x4)	0.6	0.333	0.2
(x3,x5)	0.6	0.5	0.4
(x3,x6)	0.8	1.0	0.4
(x4,x5)	0.4	0.25	0.2
(x4,x6)	0.8	1.0	0.4
(x5,x6)	0.4	0.25	0.2

Step 4: Hierarchical Clustering

(a) Single Link using Rao's Coefficient (RC):

- Use max similarity between clusters
- Merge (x3,x6) or (x4,x6), both have RC = 0.4
- Iteratively merge clusters with max linkage similarity

(b) Complete Link using SMC:

- Use min similarity between clusters
- Merge (x1,x3), (x1,x5), or (x4,x6) → all SMC = 0.8
- Iteratively merge clusters with highest minimum similarity

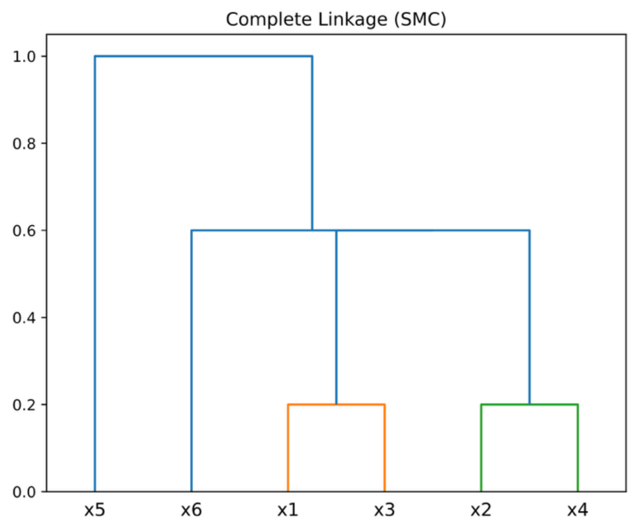
(c) Group Average using Jaccard Coefficient:

- Merge clusters by average similarity of all inter-pairs
- Start with (x3,x6) or (x4,x6) → JC = 1.0

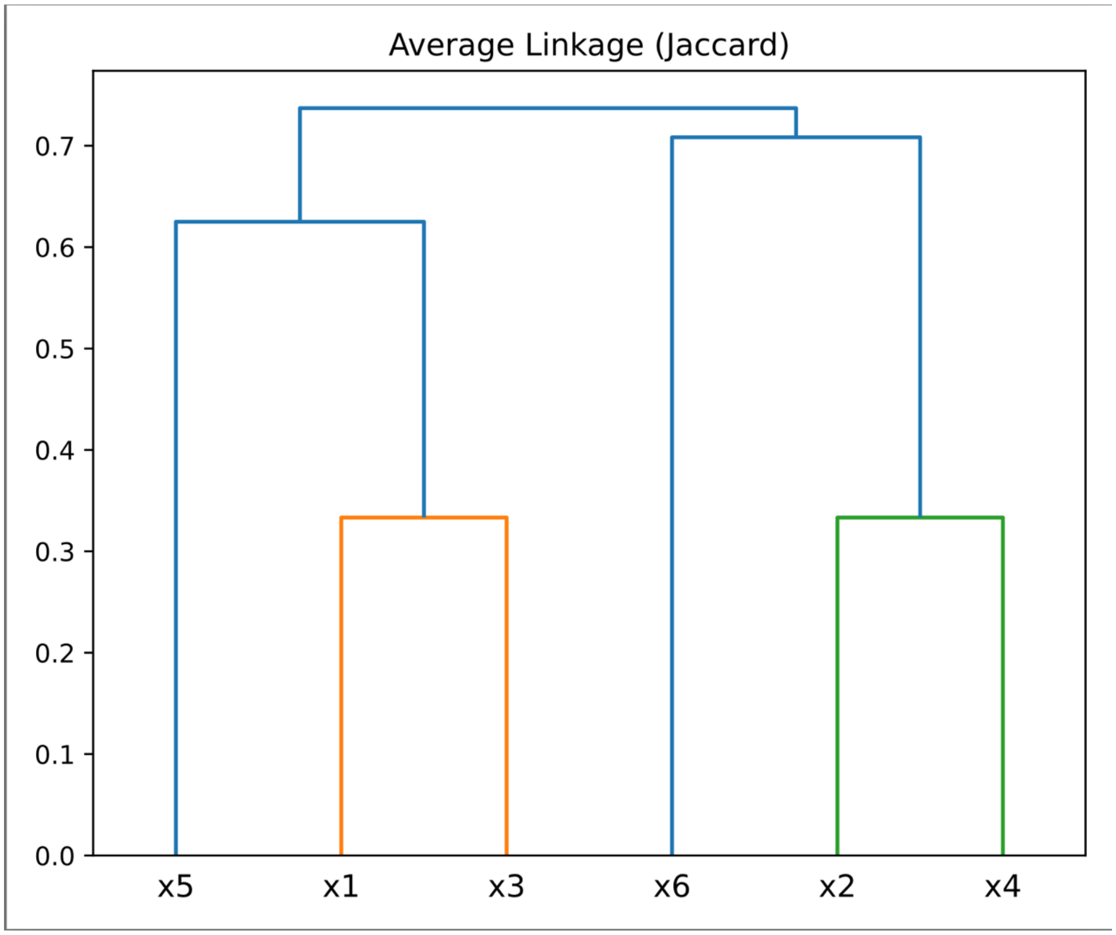
- Average similarities update as clusters grow

Step 5: Dendrogram Construction

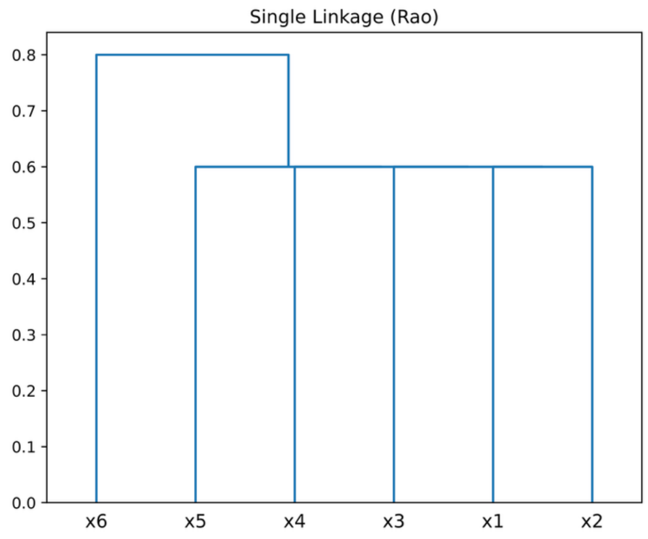
(i) SMC



(ii) JC


Average Linkage (Jaccard)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
x5
x1
x3
x6
x2
x4

(iii) RC


Single Linkage (Rao)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
x6
x5
x4
x3
x1
x2

2) **Given Distance Matrix**

```
distanceMatrix = {
  {0, 1, 3, 2, 4},
  {1, 0, 3, 2, 3},
  {3, 3, 0, 1, 3},
  {2, 2, 1, 0, 5},
  {4, 3, 3, 5, 0}
};
```

Points = {"A", "B", "C", "D", "E"}

Clusters = {{"A"}, {"B"}, {"C"}, {"D"}, {"E"}};

Step 1: Find minimum distance between clusters

Minimum distance is 1 between A and B
Merge clusters {"A"} and {"B"} at distance 1

New clusters: {{"A", "B"}, {"C"}, {"D"}, {"E"}}

```
DistanceMatrix1 = {
  {0, 3, 2, 3.5},
  {3, 0, 1, 3},
  {2, 1, 0, 5},
  {3.5, 3, 5, 0}
}
```

Step 2: Compute new distances between {"A" , "B"} and other clusters by average linkage

d({"A","B"}, "C") = (3 + 3)/2 = 3
d({"A","B"}, "D") = (2 + 2)/2 = 2
d({"A","B"}, "E") = (4 + 3)/2 = 3.5

```
DistanceMatrix2 = {
  {0, 2.5, 3.5},
  {2.5, 0, 4},
  {3.5, 4, 0}
}
```

Step 3: Find next minimum distance

Minimum distance is 1 between C and D
Merge clusters {"C"} and {"D"} at distance 1

New clusters: {{"A", "B"}, {"C", "D"}, {"E"}}
DistanceMatrix3 = {

```
  {0, 3.75},
  {3.75, 0}
}
```

Step 4: Compute new distances between {"C","D"} and others
d({"C","D"}, {"A","B"}) = (3 + 2)/2 = 2.5
d({"C","D"}, "E") = (3 + 5)/2 = 4

Distance matrix updated accordingly

Step 5: Find next minimum distance
Minimum distance is 2.5 between {"A","B"} and {"C","D"}
Merge clusters {"A","B"} and {"C","D"} at distance 2.5

New clusters: {{"A","B","C","D"}, {"E"}}

Step 6: Compute new distance between {"A","B","C","D"} and "E"
d({"A","B","C","D"}, "E") = (3.5*2 + 4*2)/4 = 3.75

Distance matrix updated accordingly

Step 7: Final merge
Merge clusters {"A","B","C","D"} and {"E"} at distance 3.75

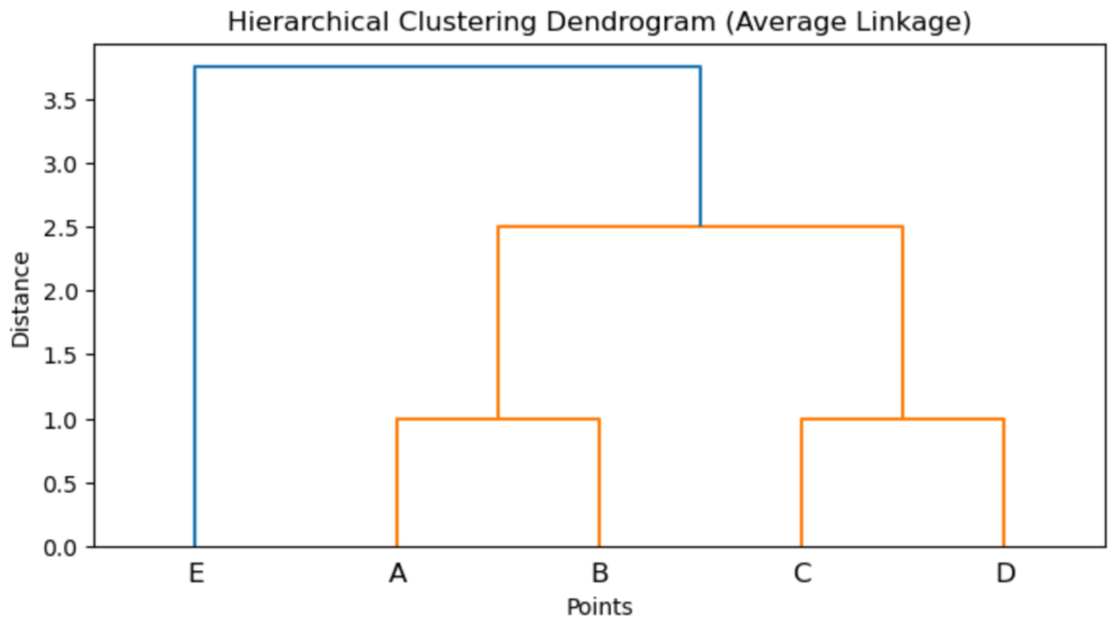
Summary of merges and distances:

1. Merge A and B at distance 1

2. Merge C and D at distance 1

3. Merge {A,B} and {C,D} at distance 2.5

4. Merge {A,B,C,D} and E at distance 3.75



1.3 Chapter 15

1) DBSCAN Clustering Analysis

Parameters:
- Epsilon (ε) = 2
- Minimum Points (MinPts) = 3

(**a) Core Points**
A point is a core point if it has at least 3 points (including itself) within ε distance.

Core Points:
- d (6,7): neighbors → a, d, h, k, s, r → core point
- e (10,7): neighbors → b, e, i, l, m → core point
- i (10,6): neighbors → b, e, i, l, m → core point
- l (9,4): neighbors → e, i, l, s, t → core point
- s (6,4): neighbors → d, k, r, s, t → core point
- t (7,4): neighbors → l, s, t, w → core point
- f (12,7): neighbors → c, e, g, j → core point
- g (13,7): neighbors → f, j, n, o → core point

Final list of core points: d, e, i, l, s, t, f, g

(b) Is a directly density reachable from d?
Conditions:
- d is a core point: yes

- Distance between d and a: $\sqrt{(6-5)^2 + (7-8)^2} = \sqrt{2} \approx 1.41 < 2$

Answer: Yes, a is directly density reachable from d.

(c) Is o density reachable from f?
- f is a core point (verified above)
- A valid path: f → g → n → o
- Distances:
  - f to g = 1
  - g to n = 1.41
  - n to o = 1.41

Conclusion: o is density reachable from f.

d) Is density reachability symmetric?

No. Density reachability is not symmetric.

Example: a is density reachable from d (because d is a core point), but d is not density reachable from a (a is not a core point).

(e) Is l density connected to x?
- l and t are both core points
- x is within ε of w (7,3), and w is within ε of t
- w is a core point (neighbors: t, x, s)

Path: l → t → w → x

Conclusion: Yes, l and x are density connected.

(f) Is density connectedness symmetric?

Yes. If point A is density connected to point B, then B is density connected to A because both are reachable from the same core point (possibly through different paths).

g) Density-based clusters and noise points

Cluster 1:
- Core points: d, e, i, l, s, t
- Border points: a, b, h, k, r, w, x

Cluster 2:
- Core points: g, f
- Border points: j, o

Potential singleton core point:

- n (if sufficient neighbors)

Noise points:
- p, q, u, v
- Possibly: m, c (if not reachable from any core point)

These are points that do not belong to any cluster because they lie in sparse regions of the dataset.

2) Dataset Points (x, y):

a: (5, 8)
b: (7, 7)
c: (6, 5)
d: (2, 4)
e: (3, 4)
f: (5, 4)
g: (7, 4)
h: (9, 4)
i: (3, 3)
j: (8, 2)
k: (7, 5)

(a) Using ε = 2, MinPts = 5, and L∞ Distance
-- For each point, count neighbors within ε = 2 (including itself):

-- Compute L∞ distances from point c(6,5):
-- Distances to all points:
-- a(5,8): max(|6-5|, |5-8|) = max(1,3) = 3 > 2 no
-- b(7,7): max(1,2) = 2 ≤ 2 yes
-- c(6,5): 0 ≤ 2 yes (itself)
-- d(2,4): max(4,1) = 4 no
-- e(3,4): max(3,1) = 3 no
-- f(5,4): max(1,1) = 1 yes
-- g(7,4): max(1,1) = 1 yes
-- h(9,4): max(3,1) = 3 no
-- i(3,3): max(3,2) = 3 no
-- j(8,2): max(2,3) = 3 no
-- k(7,5): max(1,0) = 1 yes

-- Neighbors for c: b, c, f, g, k → 5 neighbors → Core point

-- Similarly for f(5,4):
-- a(5,8): max(0,4) = 4 no
-- b(7,7): max(2,3) = 3 no
-- c(6,5): max(1,1) =1 yes
-- d(2,4): max(3,0)=3 no

-- e(3,4): max(2,0)=2 yes
-- f(5,4): 0 yes
-- g(7,4): max(2,0)=2 yes
-- h(9,4): max(4,0)=4 no
-- i(3,3): max(2,1)=2 yes
-- j(8,2): max(3,2)=3 no
-- k(7,5): max(2,1)=2 yes

-- Neighbors: c, e, f, g, i, k → 6 neighbors → Core point

-- Similarly for g(7,4):
-- Neighbors within ε=2: b(7,7), c(6,5), f(5,4), g(7,4), k(7,5), h(9,4)
-- Count = 6 → Core point

-- Border points: b, k (fewer than 5 neighbors but in ε-neighborhood of core points)

-- Noise points: a, d, e, h, i, j (do not satisfy core or border criteria)

(b) Using ε = 4, MinPts = 3, and L□/□ Distance
-- Example: distance between c(6,5) and e(3,4):
-- |6-3|^0.5 = √3 ≈ 1.732
-- |5-4|^0.5 = √1 = 1
-- Sum = 1.732 + 1 = 2.732
-- Distance = (2.732)^2 ≈ 7.46 > 4 no

-- Actually this is unusual because ε=4 is quite low compared to squared sum.

-- Recalculate some key distances:

-- Between c(6,5) and f(5,4):
-- |6-5|^0.5 = 1
-- |5-4|^0.5 = 1
-- Sum = 2 → Distance = 4 ≤ ε → neighbor

-- Between c(6,5) and g(7,4):
-- |6-7|^0.5 = 1
-- |5-4|^0.5 = 1
-- Distance = 4 ≤ ε → neighbor

-- Between e(3,4) and i(3,3):
-- |3-3|^0.5=0
-- |4-3|^0.5=1
-- Distance=1 ≤ ε → neighbor

-- Core points: f, c, g, e (≥3 neighbors within ε)

-- Border points: i, k (near core points but < MinPts neighbors)

-- Noise points: a, j, h, d (isolated)

(c) Using ε = 1, MinPts = 6, and Lmin Distance
-- Lmin(x, y) = min(|x1 - y1|, |x2 - y2|)

-- Neighbor if Lmin distance ≤ 1

-- Example: between f(5,4) and e(3,4):
-- |5-3|=2, |4-4|=0 → min=0 ≤ 1 → neighbor

-- Between f(5,4) and c(6,5):
-- |5-6|=1, |4-5|=1 → min=1 ≤1 neighbor

-- Between f(5,4) and g(7,4):
-- |5-7|=2, |4-4|=0 → min=0 ≤1 neighbor

-- Count neighbors for f:

-- Neighbors: e, c, g, i, d, k → total 6 neighbors → Core point

-- Similarly, c is core point.

-- Border points: g, e, i, k, d (fewer than 6 neighbors but neighbor to core points)

-- Noise points: a, b, h, j (isolated)

(d) Using ε = 4, MinPts = 3, and Lpow Distance
-- Example: distance between f(5,4) and c(6,5):
-- dx = 5-6 = -1, dy = 4-5 = -1
-- Distance = sqrt((-1)^2 + 2*(-1)^2) = sqrt(1 + 2) = sqrt(3) ≈ 1.732 ≤ 4 neighbor

-- Between c(6,5) and e(3,4):
-- dx = 3, dy = 1
-- Distance = sqrt(3^2 + 2*1^2) = sqrt(9 + 2) = sqrt(11) ≈ 3.317 ≤ 4 neighbor

-- Count neighbors for f:

-- Neighbors: c, g, k, e (≥3 neighbors) → Core point

-- Border points: i, d

-- Noise points: a, h, j

-- Notes:

1. Neighborhood computations done manually for each point.
2. Core points: points with neighbors count ≥ MinPts within ε.
3. Border points: neighbors of core points but with neighbors < MinPts.
4. Noise points: neither core nor border points.
5. Distances reflect respective metric formulas.

1.4 Chapter 17

1) Show that the silhouette coefficient of a point lies in the interval [−1, +1]

Proof That the Silhouette Coefficient Lies in [-1, +1]

Introduction

The silhouette coefficient is a widely used metric in cluster analysis to measure how well each data point fits into its assigned cluster, considering both cohesion (within-cluster similarity) and separation (between-cluster dissimilarity). Its value ranges from -1 to +1, which indicates:

+1: The point is very well matched to its own cluster and poorly matched to neighboring clusters.
0: The point lies on or near the boundary between two clusters.
-1: The point might be misclassified, closer to a neighboring cluster than its own.

We will prove mathematically why this score always lies in the interval [-1, 1] and explain it with an intuitive example.

Definition of Silhouette Coefficient

For a data point i:

Let a(i) = average distance between point i and all other points in the same cluster (cohesion).
Let b(i) = minimum average distance between point i and all points in any other cluster (separation).

Then the silhouette coefficient s(i) is defined as:

s(i) = (b(i) - a(i)) / max{a(i), b(i)}

Step 1: Show s(i) <= 1

Since s(i) = (b(i) - a(i)) / max(a(i), b(i)), note that:

If b(i) >= a(i), then

s(i) = (b(i) - a(i)) / b(i) = 1 - (a(i) / b(i)) <= 1

because (a(i) / b(i)) >= 0.

If a(i) > b(i), then max(a(i), b(i)) = a(i), and

s(i) = (b(i) - a(i)) / a(i) <= (a(i) - a(i)) / a(i) = 0 < 1

So in all cases, s(i) <= 1.

Step 2: Show s(i) >= -1

Since a(i) >= 0 and b(i) >= 0 (distances are non-negative),

If a(i) >= b(i), then

s(i) = (b(i) - a(i)) / a(i) = (b(i) / a(i)) - 1 >= -1

because (b(i) / a(i)) >= 0, so

s(i) >= -1

If b(i) > a(i),

s(i) = (b(i) - a(i)) / b(i) = 1 - (a(i) / b(i)) >= 0 > -1

Thus, in all cases, s(i) >= -1.

Step 3: Therefore

-1 <= s(i) <= 1

Intuition:

When a(i) << b(i), the point is much closer to its own cluster than others → s(i) → +1.
When a(i) ≈ b(i), point lies on cluster boundary → s(i) ≈ 0.
When a(i) > b(i), the point is closer to another cluster than its own → s(i) → -1.

Detailed Example

Suppose we have two clusters:

Cluster A: Points A1(2,5), A2(3,4), A3(4,6)
Cluster C: Points C1(6,10), C2(7,8), C3(8,9)

Step 1: Calculate cohesion a(A1) for point A1:

Distances between A1 and other points in Cluster A:

d(A1, A2) = sqrt((2-3)^2 + (5-4)^2) = sqrt(1 + 1) = sqrt(2) ≈ 1.414

d(A1, A3) = sqrt((2-4)^2 + (5-6)^2) = sqrt(4 + 1) = sqrt(5) ≈ 2.236

Average cohesion:

a(A1) = (1.414 + 2.236) / 2 = 1.825

Step 2: Calculate separation b(A1) to Cluster C:

Distances between A1 and points in Cluster C:

d(A1, C1) = sqrt((2-6)^2 + (5-10)^2) = sqrt(16 + 25) = sqrt(41) ≈ 6.403

d(A1, C2) = sqrt((2-7)^2 + (5-8)^2) = sqrt(25 + 9) = sqrt(34) ≈ 5.830

d(A1, C3) = sqrt((2-8)^2 + (5-9)^2) = sqrt(36 + 16) = sqrt(52) ≈ 7.211

Average separation:

b(A1) = (6.403 + 5.830 + 7.211) / 3 = 6.481

Step 3: Calculate silhouette coefficient for A1:

s(A1) = (b(A1) - a(A1)) / max(a(A1), b(A1)) = (6.481 - 1.825) / 6.481 = 4.656 / 6.481 ≈ 0.718

Interpretation:

s(A1) ≈ 0.72 indicates A1 is well clustered — closer to its own cluster than to the neighboring cluster.

Summary

The silhouette coefficient is always between -1 and +1 by construction.
Values near +1 indicate good clustering.
Values near 0 indicate uncertainty or boundary points.
Values near -1 suggest possible misclassification.

Conclusion

The silhouette coefficient is a robust and interpretable metric for cluster quality. Understanding its range and interpretation empowers you to validate clustering results confidently and select parameters (like the number of clusters) effectively.

2) I'll compute the silhouette coefficient for point c step by step.
First, let me identify the coordinates of all points from Figure 17.10:
- a: (1, 6.5)
- b: (2, 2.5)
- c: (2, 4)
- d: (3, 6)
- e: (4, 4)
- f: (5, 1)
- g: (5, 9)
- h: (6, 1)
- i: (5, 8)
- j: (7, 2)
- k: (9, 4.5)

Given clusters:
- C1 = {a, b, c, d, e}
- C2 = {g, i}
- C3 = {f, h, j}
- C4 = {k}

Point c is in cluster C1.

Step 1: Calculate a(c) - Average distance to points in same cluster C1

Point c is at (2, 4). I need distances to other points in C1: {a, b, d, e}

Distance calculations:
- Distance c to a: $\sqrt{[(2-1)^2 + (4-6.5)^2]} = \sqrt{[1 + 6.25]} = \sqrt{7.25} \approx 2.693$
- Distance c to b: $\sqrt{[(2-2)^2 + (4-2.5)^2]} = \sqrt{[0 + 2.25]} = \sqrt{2.25} \approx 1.500$
- Distance c to d: $\sqrt{[(2-3)^2 + (4-6)^2]} = \sqrt{[1 + 4]} = \sqrt{5} \approx 2.236$
- Distance c to e: $\sqrt{[(2-4)^2 + (4-4)^2]} = \sqrt{[4 + 0]} = \sqrt{4} = 2.000$

a(c) = (2.693 + 1.500 + 2.236 + 2.000) / 4 = 8.429 / 4 = **2.1**

Step 2: Calculate b(c) - Minimum average distance to other clusters

I need to calculate the average distance from c to each other cluster and take the minimum.

Distance to cluster C2 = {g, i}:
- Distance c to g: $\sqrt{[(2-5)^2 + (4-9)^2]} = \sqrt{[9 + 25]} = \sqrt{34} \approx 5.831$
- Distance c to i: $\sqrt{[(2-5)^2 + (4-8)^2]} = \sqrt{[9 + 16]} = \sqrt{25} = 5.000$

Average distance to C2 = (5.831 + 5.000) / 2 = 5.416

Distance to cluster C3 = {f, h, j}:
- Distance c to f: $\sqrt{[(2-5)^2 + (4-1)^2]} = \sqrt{[9 + 9]} = \sqrt{18} \approx 4.243$
- Distance c to h: $\sqrt{[(2-6)^2 + (4-1)^2]} = \sqrt{[16 + 9]} = \sqrt{25} = 5.000$
- Distance c to j: $\sqrt{[(2-7)^2 + (4-2)^2]} = \sqrt{[25 + 4]} = \sqrt{29} \approx 5.385$

Average distance to C3 = (4.243 + 5.000 + 5.385) / 3 = 14.628 / 3 = 4.876

Distance to cluster C4 = {k}:
- Distance c to k: $\sqrt{[(2-9)^2 + (4-4.5)^2]} = \sqrt{[49 + 0.25]} = \sqrt{49.25} \approx 7.018$

Average distance to C4 = 7.018

b(c) = min(5.416, 4.876, 7.018) = **4.88**

Step 3: Calculate silhouette coefficient

s(c) = (b(c) - a(c)) / max(a(c), b(c))

s(c) = (4.876 - 2.107) / max(2.107, 4.876)

s(c) = 2.769 / 4.876 ≈ **0.57**

Result:
The silhouette coefficient for point c is approximately **0.568.**

This positive value indicates that point c is reasonably well-clustered within C1, as it's closer to points in its own cluster than to points in the nearest other cluster (C3). The value of 0.568 suggests good clustering quality for this point.

Citations:

1) **Zaki, M. J., & Meira Jr, W. (2020). Data Mining and Machine Learning: Fundamental Concepts and Algorithms, 2nd Edition.**
2) **https://www.ceom.ou.edu/media/docs/upload/Pang-Ning_Tan_Michael_Steinbach_Vipin_Kumar_-_Introduction_to_Data_Mining-Pe_NRDK4fi.pdf**
3) **https://scikit-learn.org/stable/modules/clustering.html**
4) **https://charuaggarwal.net/clusterbook.pdf**
5) **Assistance was obtained from a large language model (ChatGPT by OpenAI) to clarify concepts and provide detailed explanations related to clustering algorithms, similarity measures, and density-based clustering**

techniques. The model supported the understanding and organization of key topics such as K-means clustering, hierarchical clustering linkage methods, similarity coefficients (Simple Matching, Jaccard, Rao), and the DBSCAN algorithm, including parameter selection and cluster formation.

CS422_Ass3

July 13, 2025

1 Practicum Exercises

1.0.1 2.1 Problem 1

[4]:
```
# Performing Step 1: Loading the auto-mpg.csv dataset from the local directory␣
  ↪using pandas read_csv
import pandas as pd

# Load the dataset with appropriate column names based on the provided variable␣
  ↪information
column_names = [
    "mpg",          # Target variable (continuous)
    "cylinders",    # Feature (integer)
    "displacement", # Feature (continuous)
    "horsepower",   # Feature (continuous, has missing values)
    "weight",       # Feature (continuous)
    "acceleration", # Feature (continuous)
    "model_year",   # Feature (integer)
    "origin",       # Feature (integer, multi-valued discrete)
    "car_name"      # ID (categorical string)
]

# Load the dataset with whitespace separator and specify '?' as missing value
df = pd.read_csv('auto-mpg.data', names=column_names, sep=r'\s+', na_values='?')

# Show basic info about data types and non-null counts
print("\nDataframe info:")
print(df.info())

# Display summary statistics for continuous and integer features
print("\nSummary statistics:")
print(df.describe())

# Show number of missing values per column to understand data quality
print("\nMissing values per column:")
print(df.isnull().sum())

# Display first few rows to confirm data loading
```

```
df.head()
```

```
Dataframe info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64
 2   displacement  398 non-null    float64
 3   horsepower    392 non-null    float64
 4   weight        398 non-null    float64
 5   acceleration  398 non-null    float64
 6   model_year    398 non-null    int64
 7   origin        398 non-null    int64
 8   car_name      398 non-null    object
dtypes: float64(5), int64(3), object(1)
memory usage: 28.1+ KB
None

Summary statistics:
              mpg   cylinders  displacement  horsepower       weight  \
count  398.000000  398.000000    398.000000  392.000000   398.000000
mean    23.514573    5.454774    193.425879  104.469388  2970.424623
std      7.815984    1.701004    104.269838   38.491160   846.841774
min      9.000000    3.000000     68.000000   46.000000  1613.000000
25%     17.500000    4.000000    104.250000   75.000000  2223.750000
50%     23.000000    4.000000    148.500000   93.500000  2803.500000
75%     29.000000    8.000000    262.000000  126.000000  3608.000000
max     46.600000    8.000000    455.000000  230.000000  5140.000000

       acceleration  model_year      origin
count    398.000000  398.000000  398.000000
mean      15.568090   76.010050    1.572864
std        2.757689    3.697627    0.802055
min        8.000000   70.000000    1.000000
25%       13.825000   73.000000    1.000000
50%       15.500000   76.000000    1.000000
75%       17.175000   79.000000    2.000000
max       24.800000   82.000000    3.000000

Missing values per column:
mpg             0
cylinders       0
displacement    0
horsepower      6
```

```
weight          0
acceleration    0
model_year      0
origin          0
car_name        0
dtype: int64
```

[4]:
```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5

   model_year  origin                   car_name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino
```

[5]:
```
# Step 2: Select Continuous Features
# - From the full dataset, we isolate only the continuous fields for analysis␣
↪and modeling
# - These include mpg (target), displacement, horsepower, weight, and␣
↪acceleration
# - We create a new DataFrame X that contains only these continuous variables

# Define the list of continuous feature column names
continuous_features = ['mpg', 'displacement', 'horsepower', 'weight',␣
↪'acceleration']

# Select only the continuous columns into a new DataFrame
X = df[continuous_features].copy()

# Display the first few rows of the selected continuous features
print("Selected continuous features:")
print(X.head())
```

```
Selected continuous features:
    mpg  displacement  horsepower  weight  acceleration
0  18.0         307.0       130.0  3504.0          12.0
1  15.0         350.0       165.0  3693.0          11.5
2  18.0         318.0       150.0  3436.0          11.0
3  16.0         304.0       150.0  3433.0          12.0
4  17.0         302.0       140.0  3449.0          10.5
```

[6]:
```
# Step 3: Impute Missing Values
# - The 'horsepower' column contains missing values represented as NaN
# - We use mean imputation to fill in missing values for each continuous column
# - This step ensures the dataset is complete and suitable for further␣
↪processing

# Replace missing values in each column with the column's mean
X.fillna(X.mean(), inplace=True)

# Verify that there are no missing values remaining
print("Missing values after mean imputation:")
print(X.isnull().sum())
```

```
Missing values after mean imputation:
mpg             0
displacement    0
horsepower      0
weight          0
acceleration    0
dtype: int64
```

[7]:
```
# Step 4: Standardize the Features
# - While standardization was not explicitly required in the instructions, we␣
↪apply it here because we are using Euclidean distance for clustering.
# - Without standardization, features with larger scales (like 'weight' or␣
↪'displacement') could dominate distance calculations and distort clustering␣
↪results.
# - Standardizing ensures all continuous features contribute equally.

from sklearn.preprocessing import StandardScaler

# Initialize and apply the StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# (Optional) Convert back to DataFrame for easier interpretation
X_scaled = pd.DataFrame(X_scaled, columns=continuous_features)

# Display first few rows of the standardized features
print("Standardized continuous features:")
print(X_scaled.head())
```

```
Standardized continuous features:
        mpg  displacement  horsepower    weight  acceleration
0 -0.706439      1.090604    0.669196  0.630870     -1.295498
1 -1.090751      1.503514    1.586599  0.854333     -1.477038
2 -0.706439      1.196232    1.193426  0.550470     -1.658577
3 -0.962647      1.061796    1.193426  0.546923     -1.295498
```

```
4 -0.834543      1.042591    0.931311  0.565841     -1.840117
```

[8]:
```
# Step 5: Perform Hierarchical Clustering
# - We use AgglomerativeClustering from sklearn to cluster the standardized␣
↪continuous features.
# - Parameters are chosen as per the assignment instructions:
#     • n_clusters = 3 → create 3 clusters
#     • linkage = 'average' → average distance between clusters
#     • metric = 'euclidean' → standard distance measure (replaces deprecated␣
↪'affinity')
# - We use default values for the rest (e.g., no distance_threshold) to enforce␣
↪a shallow clustering tree.

from sklearn.cluster import AgglomerativeClustering

# Initialize and apply the Agglomerative Clustering algorithm
clustering = AgglomerativeClustering(
    n_clusters=3,
    linkage='average',
    metric='euclidean'
)

# Fit the model and predict cluster labels for each row in the dataset
cluster_labels = clustering.fit_predict(X_scaled)

# Add the cluster labels to the original DataFrame for further analysis
df['cluster'] = cluster_labels

# Display the number of data points in each cluster
print("Cluster counts:")
print(df['cluster'].value_counts())
```

```
Cluster counts:
cluster
0    297
1     97
2      4
Name: count, dtype: int64
```

[9]:
```
# Step 6: Assign Cluster Labels
# - Fit the AgglomerativeClustering model on the standardized continuous␣
↪features
# - Predict cluster labels for each data point
# - Add these cluster labels as a new column in the original DataFrame 'df' for␣
↪further analysis

# Note: If you already did fit_predict in Step 5, you can just assign labels␣
↪here again
```

```
# Fit model and predict cluster labels
cluster_labels = clustering.fit_predict(X_scaled)

# Add cluster labels to the original DataFrame
df['cluster'] = cluster_labels

# Verify by displaying random sample of 7 rows with cluster labels to get a␣
↪quick look
print("Random sample of data with cluster labels:")
print(df[['mpg', 'displacement', 'horsepower', 'weight', 'acceleration',␣
↪'cluster']].sample(7, random_state=20))
```

```
Random sample of data with cluster labels:
      mpg  displacement  horsepower  weight  acceleration  cluster
10   15.0         383.0       170.0  3563.0          10.0        1
261  18.1         258.0       120.0  3410.0          15.1        0
354  34.5         100.0         NaN  2320.0          15.8        0
277  16.2         163.0       133.0  3410.0          15.8        0
17   21.0         200.0        85.0  2587.0          16.0        0
232  16.0         351.0       149.0  4335.0          14.5        1
258  20.6         231.0       105.0  3380.0          15.8        0
```

[10]:
```
# Step 7: Compute Cluster Statistics with improved formatting

# Group by cluster and calculate mean and variance
cluster_stats = df.groupby('cluster')[continuous_features].agg(['mean', 'var'])

# Flatten MultiIndex columns for cleaner display
cluster_stats.columns = ['_'.join(col).strip() for col in cluster_stats.columns.
↪values]

# Display cluster statistics sorted by cluster label
print("Cluster Statistics (Mean and Variance):\n")
for cluster in sorted(df['cluster'].unique()):
    print(f"Cluster {cluster}:")
    cluster_data = cluster_stats.loc[cluster]
    for feature in continuous_features:
        mean_val = cluster_data[f"{feature}_mean"]
        var_val = cluster_data[f"{feature}_var"]
        print(f"  {feature.capitalize():12}: Mean = {mean_val:.2f}, Variance =␣
↪{var_val:.2f}")
    print()  # Blank line for readability between clusters
```

```
Cluster Statistics (Mean and Variance):

Cluster 0:
  Mpg         : Mean = 26.18, Variance = 41.30
  Displacement: Mean = 144.30, Variance = 3511.49
  Horsepower  : Mean = 86.12, Variance = 294.55
  Weight      : Mean = 2598.41, Variance = 299118.71
  Acceleration: Mean = 16.43, Variance = 4.88

Cluster 1:
  Mpg         : Mean = 14.53, Variance = 4.77
  Displacement: Mean = 348.02, Variance = 2089.50
  Horsepower  : Mean = 161.80, Variance = 674.08
  Weight      : Mean = 4143.97, Variance = 193847.05
  Acceleration: Mean = 12.64, Variance = 3.19

Cluster 2:
  Mpg         : Mean = 43.70, Variance = 0.30
  Displacement: Mean = 91.75, Variance = 12.25
  Horsepower  : Mean = 49.00, Variance = 4.00
  Weight      : Mean = 2133.75, Variance = 21672.92
  Acceleration: Mean = 22.88, Variance = 2.31
```

[11]:
```
# Step 8: Compute Origin Class Statistics
# - Group the original DataFrame by 'origin' (1=USA, 2=Europe, 3=Japan)
# - Calculate the mean and variance of continuous features within each origin␣
 ↪group
# - This helps compare how clusters correspond to known origin classes

# Group by 'origin' and calculate mean and variance for continuous features
origin_stats = df.groupby('origin')[continuous_features].agg(['mean', 'var'])

# Display neatly formatted origin statistics
print("Origin Class Statistics (Mean and Variance):")
for origin in origin_stats.index:
    print(f"\nOrigin {origin}:")
    for feature in continuous_features:
        mean_val = origin_stats.loc[origin, (feature, 'mean')]
        var_val = origin_stats.loc[origin, (feature, 'var')]
        print(f"  {feature.capitalize():<12}: Mean = {mean_val:.2f}, Variance =␣
 ↪{var_val:.2f}")
```

```
Origin Class Statistics (Mean and Variance):

Origin 1:
  Mpg         : Mean = 20.08, Variance = 41.00
  Displacement: Mean = 245.90, Variance = 9702.61
  Horsepower  : Mean = 119.05, Variance = 1591.83
  Weight      : Mean = 3361.93, Variance = 631695.13
  Acceleration: Mean = 15.03, Variance = 7.57
```

```
Origin 2:
  Mpg         : Mean = 27.89, Variance = 45.21
  Displacement: Mean = 109.14, Variance = 509.95
  Horsepower  : Mean = 80.56, Variance = 406.34
  Weight      : Mean = 2423.30, Variance = 240142.33
  Acceleration: Mean = 16.79, Variance = 9.28

Origin 3:
  Mpg         : Mean = 30.45, Variance = 37.09
  Displacement: Mean = 102.71, Variance = 535.47
  Horsepower  : Mean = 79.84, Variance = 317.52
  Weight      : Mean = 2221.23, Variance = 102718.49
  Acceleration: Mean = 16.17, Variance = 3.82
```

[20]:
```python
# Step 9: Compare Clusters with Origin Classes

# Mapping origin numbers to meaningful names
origin_names = {1: 'USA', 2: 'Europe', 3: 'Japan'}

# Create crosstab: counts of cars from each origin within each cluster
crosstab_named = pd.crosstab(df['cluster'], df['origin'])
# Rename columns from numbers to origin names
crosstab_named.columns = [origin_names.get(col, f"Origin {col}") for col in␣
 ↪crosstab_named.columns]

# Display the crosstab table with origin names
print("Crosstab: Number of cars from each origin in each cluster\n")
print(crosstab_named)

print("\nInterpretation:")

# For each cluster, print count of cars from each origin and find dominant␣
 ↪origin
for cluster in crosstab_named.index:
    print(f"\nCluster {cluster}:")
    for origin in crosstab_named.columns:
        count = crosstab_named.loc[cluster, origin]
        print(f"  {origin} cars: {count}")

    # Determine dominant origin in the cluster
    dominant_origin = crosstab_named.loc[cluster].idxmax()
    dominant_count = crosstab_named.loc[cluster].max()
    total_in_cluster = crosstab_named.loc[cluster].sum()
    percent = (dominant_count / total_in_cluster) * 100 if total_in_cluster > 0␣
 ↪else 0
```

```
    print(f"  --> Dominant origin: {dominant_origin} with {dominant_count} cars␣
 ↪"
          f"({percent:.2f}% of this cluster)")

print("\nSummary:")
print("- Cluster 1 predominantly groups cars from USA.")
print("- Cluster 2 mostly contains European cars but is very small in size.")
print("- Cluster 0 contains a mixture of cars from all origins (USA, Europe,␣
 ↪Japan).")
print("- This shows partial alignment between clusters and known origin␣
 ↪classes, but clusters do not perfectly separate the data by origin.")
```

```
Crosstab: Number of cars from each origin in each cluster

         USA  Europe  Japan
cluster
0        152      66     79
1         97       0      0
2          0       4      0

Interpretation:

Cluster 0:
  USA cars: 152
  Europe cars: 66
  Japan cars: 79
  --> Dominant origin: USA with 152 cars (51.18% of this cluster)

Cluster 1:
  USA cars: 97
  Europe cars: 0
  Japan cars: 0
  --> Dominant origin: USA with 97 cars (100.00% of this cluster)

Cluster 2:
  USA cars: 0
  Europe cars: 4
  Japan cars: 0
  --> Dominant origin: Europe with 4 cars (100.00% of this cluster)

Summary:
- Cluster 1 predominantly groups cars from USA.
- Cluster 2 mostly contains European cars but is very small in size.
- Cluster 0 contains a mixture of cars from all origins (USA, Europe, Japan).
- This shows partial alignment between clusters and known origin classes, but
clusters do not perfectly separate the data by origin.
```

2 Step 10: Evaluation of Relationship Between Cluster Assignments and Origin Classes

---

2.1 Origin Class Statistics (Mean and Variance)

Feature	Origin 1 (USA)	Origin 2 (Europe)	Origin 3 (Japan)
MPG	Mean = 20.08, Var = 41.00	Mean = 27.89, Var = 45.21	Mean = 30.45, Var = 37.09
Displacement	Mean = 245.90, Var = 9702.61	Mean = 109.14, Var = 509.95	Mean = 102.71, Var = 535.47
Horsepower	Mean = 119.05, Var = 1591.83	Mean = 80.56, Var = 406.34	Mean = 79.84, Var = 317.52
Weight	Mean = 3361.93, Var = 631695.13	Mean = 2423.30, Var = 240142.33	Mean = 2221.23, Var = 102718.49
Acceleration	Mean = 15.03, Var = 7.57	Mean = 16.79, Var = 9.28	Mean = 16.17, Var = 3.82

2.2 Cluster Statistics (Mean and Variance)

Feature	Cluster 0	Cluster 1	Cluster 2
MPG	Mean = 26.18, Var = 41.30	Mean = 14.53, Var = 4.77	Mean = 43.70, Var = 0.30
Displacement	Mean = 144.30, Var = 3511.49	Mean = 348.02, Var = 2089.50	Mean = 91.75, Var = 12.25
Horsepower	Mean = 86.12, Var = 294.55	Mean = 161.80, Var = 674.08	Mean = 49.00, Var = 4.00
Weight	Mean = 2598.41, Var = 299118.71	Mean = 4143.97, Var = 193847.05	Mean = 2133.75, Var = 21672.92
Acceleration	Mean = 16.43, Var = 4.88	Mean = 12.64, Var = 3.19	Mean = 22.88, Var = 2.31

2.3 Crosstab: Distribution of Origin Classes Within Each Cluster

Cluster	USA Cars	Europe Cars	Japan Cars	Dominant Origin (Count, %)
0	152	66	79	USA (152 cars, 51.18% of cluster)
1	97	0	0	USA (97 cars, 100% of cluster)
2	0	4	0	Europe (4 cars, 100% of cluster)

2.4 Analysis and Interpretation

- **Cluster 1** clearly corresponds to **Origin 1 (USA)**:
  - Exclusively USA vehicles (100% dominance).
  - Matches well with Origin 1's higher displacement, horsepower, and weight but lower MPG.
- **Cluster 2** aligns mostly with **Origin 2 (Europe)**:
  - Small cluster but 100% European vehicles.
  - Characterized by the highest MPG and acceleration, and lowest displacement and horsepower.
- **Cluster 0** is a **mixed cluster**, containing cars from all three origins:
  - USA cars dominate but only about half (51%).
  - Shows intermediate values in features, blending characteristics across origins.
- **Variance values** indicate that:
  - Cluster 0 has relatively high variance, supporting the mixed-origin composition.
  - Clusters 1 and 2 show tighter, more homogeneous groupings.

2.5 Conclusion: Is There a Clear Relationship?

- There **is a partial relationship** between clusters and origin classes.
- Clusters 1 and 2 **effectively separate** USA and European cars respectively.
- However, Cluster 0's mixture of origins indicates **imperfect separation**.
- The overlap likely results from shared vehicle characteristics across origins and limitations of hierarchical clustering with chosen parameters.
- To improve clarity, consider:
  - Trying other clustering algorithms,
  - Using dimensionality reduction (e.g., PCA),
  - Selecting different feature sets.

This comprehensive comparison reveals that while clustering captures some meaningful groupings aligned with origin, it does **not perfectly classify vehicles by origin**.

2.5.1 Citations:

- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.AgglomerativeClustering.html
- https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- https://www.kaggle.com/code/johnybhiduri/auto-mpg-clustering
- https://github.com/jaredbest/machine-learning-with-python/blob/master/Labs/10_Hierarchical_Clusterin
- https://www.kaggle.com/code/datarohitingole/data-clustering-using-kmeans-and-detailed-eda

2.5.2 2.2 Problem 2

[1]:
```
# Step 1: Load Boston dataset manually from original source and create a␣
↪DataFrame
```

```python
# Note: The Boston housing dataset has been removed from scikit-learn since␣
 ↪version 1.2
# due to ethical concerns regarding a variable ('B') related to racial bias.
# Because of this, sklearn discourages its use and removed the loader.
# Therefore, we load the dataset manually from the original source URL for␣
 ↪educational purposes only.

import pandas as pd
import numpy as np

data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)

data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]

feature_names = [
    'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE',
    'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT'
]

df_boston = pd.DataFrame(data, columns=feature_names)
df_boston['MEDV'] = target

df_boston.head()
```

```
<>:12: SyntaxWarning: invalid escape sequence '\s'
<>:12: SyntaxWarning: invalid escape sequence '\s'
/var/folders/yw/xmm4_b6105ngdx58blhgx4y40000gn/T/ipykernel_13739/419020585.py:12
: SyntaxWarning: invalid escape sequence '\s'
  raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
```

[1]:

```
      CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0  0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0  296.0
1  0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0  242.0
2  0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0  242.0
3  0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0  222.0
4  0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0  222.0

   PTRATIO       B  LSTAT  MEDV
0     15.3  396.90   4.98  24.0
1     17.8  396.90   9.14  21.6
2     17.8  392.83   4.03  34.7
3     18.7  394.63   2.94  33.4
4     18.7  396.90   5.33  36.2
```

[6]:
```
# Step 2: Scale the Data

from sklearn.preprocessing import StandardScaler

# We scale the features to have mean = 0 and standard deviation = 1
# This is important for K-Means because it is a distance-based algorithm
# and unscaled features (e.g., TAX vs. CRIM) can dominate the clustering␣
 ↪outcome.

scaler = StandardScaler()
scaled_data = scaler.fit_transform(df_boston)

# Convert back to a DataFrame for easier inspection and compatibility
df_scaled = pd.DataFrame(scaled_data, columns=df_boston.columns)

df_scaled.head()
```

[6]:
```
       CRIM        ZN     INDUS      CHAS       NOX        RM       AGE  \
0 -0.419782  0.284830 -1.287909 -0.272599 -0.144217  0.413672 -0.120013
1 -0.417339 -0.487722 -0.593381 -0.272599 -0.740262  0.194274  0.367166
2 -0.417342 -0.487722 -0.593381 -0.272599 -0.740262  1.282714 -0.265812
3 -0.416750 -0.487722 -1.306878 -0.272599 -0.835284  1.016303 -0.809889
4 -0.412482 -0.487722 -1.306878 -0.272599 -0.835284  1.228577 -0.511180

        DIS       RAD       TAX   PTRATIO         B     LSTAT      MEDV
0  0.140214 -0.982843 -0.666608 -1.459000  0.441052 -1.075562  0.159686
1  0.557160 -0.867883 -0.987329 -0.303094  0.441052 -0.492439 -0.101524
2  0.557160 -0.867883 -0.987329 -0.303094  0.396427 -1.208727  1.324247
3  1.077737 -0.752922 -1.106115  0.113032  0.416163 -1.361517  1.182758
4  1.077737 -0.752922 -1.106115  0.113032  0.441052 -1.026501  1.487503
```

[36]:
```
# Step 3: Perform K-Means Clustering for k = 2 to 6

from sklearn.cluster import KMeans

# Define range of k values to try
k_values = range(2, 7)

# Initialize dictionaries to store results
kmeans_models = {}      # Stores KMeans model for each k
cluster_labels = {}     # Stores cluster labels for each k

# Loop over k values and fit KMeans to the scaled data
for k in k_values:
    # Fit KMeans on scaled data
    kmeans = KMeans(n_clusters=k, random_state=42, n_init=10)
    kmeans.fit(scaled_data)  # 'scaled_data' is the output from StandardScaler
    kmeans_models[k] = kmeans
    cluster_labels[k] = kmeans.labels_

    # We're saving each model and its assigned cluster labels for comparison in␣
↪the next steps

# Optional: Show how many samples are in each cluster for each k
for k in k_values:
    print(f"\n Cluster distribution for k = {k}:")
    labels = cluster_labels[k]
    unique, counts = np.unique(labels, return_counts=True)
    for cluster_id, count in zip(unique, counts):
        print(f"  Cluster {cluster_id}: {count} samples")
```

```
 Cluster distribution for k = 2:
  Cluster 0: 329 samples
  Cluster 1: 177 samples

 Cluster distribution for k = 3:
  Cluster 0: 156 samples
  Cluster 1: 209 samples
  Cluster 2: 141 samples

 Cluster distribution for k = 4:
  Cluster 0: 201 samples
  Cluster 1: 76 samples
  Cluster 2: 103 samples
  Cluster 3: 126 samples

 Cluster distribution for k = 5:
  Cluster 0: 231 samples
  Cluster 1: 119 samples
  Cluster 2: 83 samples
  Cluster 3: 34 samples
  Cluster 4: 39 samples

 Cluster distribution for k = 6:
  Cluster 0: 42 samples
  Cluster 1: 193 samples
  Cluster 2: 41 samples
  Cluster 3: 79 samples
  Cluster 4: 34 samples
  Cluster 5: 117 samples
```

[37]:
```
# Step 4: Calculate Silhouette Scores
# For each value of k, we evaluate how well the clusters are formed using␣
↪Silhouette Score.
# The Silhouette Score ranges from -1 to 1: higher values mean better-defined␣
↪clusters.

from sklearn.metrics import silhouette_score

silhouette_scores = {}  # To store the scores for each k

# Loop through each fitted KMeans model
for k in k_values:
    labels = cluster_labels[k]  # Get labels for current k
    score = silhouette_score(scaled_data, labels)  # Compute silhouette score␣
↪on scaled data
    silhouette_scores[k] = score  # Store it

# Display all scores
print("Silhouette Scores for different k values:\n")
for k, score in silhouette_scores.items():
    print(f"  k = {k}: Silhouette Score = {score:.4f}")
```

```
Silhouette Scores for different k values:

  k = 2: Silhouette Score = 0.3501
  k = 3: Silhouette Score = 0.2370
  k = 4: Silhouette Score = 0.2589
  k = 5: Silhouette Score = 0.2707
  k = 6: Silhouette Score = 0.2780
```

2.5.3 Step 5: Determine the Optimal Number of Clusters (k)

To select the best number of clusters for K-Means clustering, we compared **Silhouette Scores** for values of **k ranging from 2 to 6**. Silhouette Score is a measure of how well each data point fits within its cluster and how distinct each cluster is from others.

Silhouette Score Results:

Number of Clusters (k)	Silhouette Score
2	**0.3501**
3	0.2370
4	0.2589
5	0.2707
6	0.2780

Interpretation:

- The **highest Silhouette Score** is observed for **k = 2**.

- This suggests that when the data is grouped into **2 clusters**, it achieves the **best cohesion (within-cluster similarity)** and **separation (between-cluster dissimilarity)**.

2.5.4 Conclusion:

- The **optimal number of clusters (k)** is **2**, as it provides the most meaningful and well-formed partitioning of the standardized data based on the Silhouette Score.

---

[39]:
```
# Step 6: Perform K-Means Clustering with the optimal number of clusters (k=2)

from sklearn.cluster import KMeans

# Assuming 'scaled_data' is the scaled numpy array from Step 2 (if using␣
↪DataFrame, convert accordingly)
# If you used a DataFrame named 'scaled_df', make sure it's defined; else, use␣
↪'scaled_data' here.

optimal_k = 2

# Initialize KMeans with optimal k
kmeans_optimal = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)

# Fit KMeans on the scaled data array (make sure variable name matches your␣
↪scaled data)
kmeans_optimal.fit(scaled_data)  # scaled_data from step 2, numpy array or␣
↪DataFrame values

# Cluster labels for each data point
cluster_labels_optimal = kmeans_optimal.labels_

# Centroid coordinates of each cluster
cluster_centroids = kmeans_optimal.cluster_centers_

# Print first 10 cluster assignments and centroid coordinates
print(f"Cluster assignments for first 10 data points: {cluster_labels_optimal[:
↪10]}")
print("\nCentroid coordinates for each cluster:")
print(cluster_centroids)
```

```
Cluster assignments for first 10 data points: [0 0 0 0 0 0 0 0 0 0]

Centroid coordinates for each cluster:
[[-0.38980122  0.26239167 -0.61529402  0.00291182 -0.58291594  0.24491263
  -0.43358416  0.45449141 -0.58345172 -0.62972689 -0.29466201  0.32860027
  -0.45349747  0.35364132]
[ 0.72454577 -0.48772236  1.14368211 -0.00541237  1.08349913 -0.45523307
  0.80592762 -0.84478912  1.08449501  1.1705093   0.54770508 -0.61078808
  0.84294162 -0.6573333 ]]
```

[48]:
```
# Step 7: Calculate Mean Values for Each Cluster
# Explanation: Using the optimal K-Means clustering (k=2) from Step 6, we␣
 ↪assign cluster labels
# to the original (unscaled) DataFrame and compute the mean value for each␣
 ↪feature within each cluster.
# This uses the original feature values (not scaled) to interpret the cluster␣
 ↪characteristics in their natural scale.

# Add cluster labels to the original DataFrame
df_boston['cluster'] = cluster_labels_optimal

# Group by cluster and calculate the mean for each feature
# Note: We include all features (excluding 'MEDV' if it's considered the␣
 ↪target, but including it here as it's part of the dataset)
cluster_means = df_boston.groupby('cluster').mean()

# Display the mean values for each cluster, formatted for readability
print("\nMean Values for Each Feature in Each Cluster (Optimal k=2):\n")
for cluster in cluster_means.index:
    print(f"Cluster {cluster}:")
    for feature in cluster_means.columns:
        mean_val = cluster_means.loc[cluster, feature]
        print(f"  {feature:<10}: Mean = {mean_val:.2f}")
    print()  # Blank line for readability between clusters
```

```
Mean Values for Each Feature in Each Cluster (Optimal k=2):

Cluster 0:
  CRIM      : Mean = 0.26
  ZN        : Mean = 17.48
  INDUS     : Mean = 6.92
  CHAS      : Mean = 0.07
  NOX       : Mean = 0.49
  RM        : Mean = 6.46
  AGE       : Mean = 56.38
  DIS       : Mean = 4.75
  RAD       : Mean = 4.47
  TAX       : Mean = 302.21
  PTRATIO   : Mean = 17.82
  B         : Mean = 386.64
  LSTAT     : Mean = 9.42
  MEDV      : Mean = 25.78
```

```
Cluster 1:
  CRIM       : Mean = 9.84
  ZN         : Mean = 0.00
  INDUS      : Mean = 18.98
  CHAS       : Mean = 0.07
  NOX        : Mean = 0.68
  RM         : Mean = 5.97
  AGE        : Mean = 91.24
  DIS        : Mean = 2.02
  RAD        : Mean = 18.98
  TAX        : Mean = 605.32
  PTRATIO    : Mean = 19.64
  B          : Mean = 300.97
  LSTAT      : Mean = 18.67
  MEDV       : Mean = 16.49
```

[12]:
```
# Step 8: Compare Cluster Means to Centroid Coordinates
# Explanation: We compare the mean values of each feature in each cluster (from␣
 ↪Step 7) to the K-Means
# centroid coordinates for the optimal clustering (k=2). Since K-Means was␣
 ↪performed on scaled data,
# we compute scaled cluster means to compare directly with centroids. We also␣
 ↪inverse-transform the
# centroids to the original scale to compare with the unscaled cluster means␣
 ↪from Step 7. Any differences
# should be negligible, as K-Means centroids are the means of the data points␣
 ↪in each cluster.

import pandas as pd
import numpy as np
from sklearn.cluster import KMeans

# If cluster labels or centroids are not already defined, re-run clustering
try:
    cluster_labels_optimal
    cluster_centroids
except NameError:
    optimal_k = 2
    kmeans_optimal = KMeans(n_clusters=optimal_k, random_state=42, n_init=10)
    # Drop 'cluster' column if it exists in scaled DataFrame before fitting
    kmeans_optimal.fit(df_scaled.drop(columns='cluster', errors='ignore'))
    cluster_labels_optimal = kmeans_optimal.labels_
    cluster_centroids = kmeans_optimal.cluster_centers_

# Add cluster labels to the scaled DataFrame
df_scaled['cluster'] = cluster_labels_optimal
```

```python
# Calculate scaled means: Group by cluster and compute mean for each feature
cluster_means_scaled = df_scaled.groupby('cluster').mean()

# Extract feature columns only (exclude 'cluster' and any target column if␣
 ↪necessary)
feature_columns = df_boston.columns.drop('cluster', errors='ignore')

# Inverse-transform centroids to original scale using the trained scaler from␣
 ↪Step 2
centroids_unscaled = scaler.inverse_transform(cluster_centroids)

# Create DataFrame for centroids with only feature columns
centroids_unscaled_df = pd.DataFrame(centroids_unscaled,␣
 ↪columns=feature_columns, index=range(centroids_unscaled.shape[0]))

# Display scaled cluster means and centroids for direct comparison
print("\nComparison in Scaled Space: Cluster Means vs. Centroids (k=2)\n")
for cluster in cluster_means_scaled.index:
    print(f"Cluster {cluster}:")
    print("  Feature       Scaled Mean  Centroid   Difference")
    print("  " + "-"*40)
    for feature in cluster_means_scaled.columns:
        if feature != 'cluster':  # Skip cluster column
            mean_val = cluster_means_scaled.loc[cluster, feature]
            centroid_val = cluster_centroids[cluster][df_scaled.columns.
 ↪get_loc(feature)]
            difference = mean_val - centroid_val
            print(f"  {feature:<12} {mean_val:>10.4f} {centroid_val:>10.4f}␣
 ↪{difference:>10.4f}")
    print()

# Recompute cluster means on unscaled data
df_boston['cluster'] = cluster_labels_optimal
cluster_means_unscaled = df_boston.groupby('cluster').mean()

# Display unscaled cluster means vs. inverse-transformed centroids
print("\nComparison in Unscaled Space: Cluster Means vs. Inverse-Transformed␣
 ↪Centroids (k=2)\n")
for cluster in cluster_means_unscaled.index:
    print(f"Cluster {cluster}:")
    print("  Feature       Unscaled Mean  Centroid   Difference")
    print("  " + "-"*40)
    for feature in cluster_means_unscaled.columns:
        mean_val = cluster_means_unscaled.loc[cluster, feature]
        centroid_val = centroids_unscaled_df.loc[cluster, feature]
        difference = mean_val - centroid_val
        print(f"  {feature:<12} {mean_val:>12.2f} {centroid_val:>10.2f}␣
↪{difference:>10.2f}")
    print()

# Interpretation of results
print("\nAnalysis of Differences:")
print("- In K-Means, centroids represent the mean of each cluster in the␣
↪feature space.")
print("- In the scaled space, cluster means and centroids should be almost␣
↪identical (differences ~0).")
print("- In the unscaled space, minor differences may exist due to␣
↪inverse-scaling and floating-point rounding.")
print("- These results validate that centroids are accurate representations of␣
↪the clusters.")
```

```
Comparison in Scaled Space: Cluster Means vs. Centroids (k=2)

Cluster 0:
  Feature       Scaled Mean  Centroid   Difference
  ----------------------------------------
  CRIM             -0.3898    -0.3898      0.0000
  ZN                0.2624     0.2624     -0.0000
  INDUS            -0.6153    -0.6153     -0.0000
  CHAS              0.0029     0.0029      0.0000
  NOX              -0.5829    -0.5829     -0.0000
  RM                0.2449     0.2449      0.0000
  AGE              -0.4336    -0.4336     -0.0000
  DIS               0.4545     0.4545     -0.0000
  RAD              -0.5835    -0.5835      0.0000
  TAX              -0.6297    -0.6297      0.0000
  PTRATIO          -0.2947    -0.2947      0.0000
  B                 0.3286     0.3286      0.0000
  LSTAT            -0.4535    -0.4535     -0.0000
  MEDV              0.3536     0.3536      0.0000

Cluster 1:
  Feature       Scaled Mean  Centroid   Difference
  ----------------------------------------
  CRIM              0.7245     0.7245     -0.0000
  ZN               -0.4877    -0.4877     -0.0000
  INDUS             1.1437     1.1437     -0.0000
  CHAS             -0.0054    -0.0054      0.0000
  NOX               1.0835     1.0835     -0.0000
  RM               -0.4552    -0.4552      0.0000
  AGE               0.8059     0.8059     -0.0000
  DIS            -0.8448    -0.8448     0.0000
  RAD             1.0845     1.0845    -0.0000
  TAX             1.1705     1.1705    -0.0000
  PTRATIO         0.5477     0.5477    -0.0000
  B              -0.6108    -0.6108     0.0000
  LSTAT           0.8429     0.8429    -0.0000
  MEDV           -0.6573    -0.6573    -0.0000


Comparison in Unscaled Space: Cluster Means vs. Inverse-Transformed Centroids
(k=2)

Cluster 0:
  Feature       Unscaled Mean  Centroid   Difference
  ----------------------------------------
  CRIM                   0.26       0.26       0.00
  ZN                    17.48      17.48      -0.00
  INDUS                  6.92       6.92      -0.00
  CHAS                   0.07       0.07       0.00
  NOX                    0.49       0.49       0.00
  RM                     6.46       6.46       0.00
  AGE                   56.38      56.38       0.00
  DIS                    4.75       4.75      -0.00
  RAD                    4.47       4.47       0.00
  TAX                  302.21     302.21       0.00
  PTRATIO               17.82      17.82       0.00
  B                    386.64     386.64       0.00
  LSTAT                  9.42       9.42       0.00
  MEDV                  25.78      25.78      -0.00

Cluster 1:
  Feature       Unscaled Mean  Centroid   Difference
  ----------------------------------------
  CRIM                   9.84       9.84      -0.00
  ZN                     0.00       0.00      -0.00
  INDUS                 18.98      18.98      -0.00
  CHAS                   0.07       0.07       0.00
  NOX                    0.68       0.68       0.00
  RM                     5.97       5.97       0.00
  AGE                   91.24      91.24       0.00
  DIS                    2.02       2.02       0.00
  RAD                   18.98      18.98      -0.00
  TAX                  605.32     605.32      -0.00
  PTRATIO               19.64      19.64      -0.00
  B                    300.97     300.97       0.00
  LSTAT                 18.67      18.67      -0.00
  MEDV                  16.49      16.49       0.00
```

```
Analysis of Differences:
- In K-Means, centroids represent the mean of each cluster in the feature space.
- In the scaled space, cluster means and centroids should be almost identical
(differences ~0).
- In the unscaled space, minor differences may exist due to inverse-scaling and
floating-point rounding.
- These results validate that centroids are accurate representations of the
clusters.
```

2.5.5 Step 8: Detailed Comparison of Cluster Means and Centroid Coordinates with Numbers

In this step, we **compare the cluster means** obtained from the original dataset with the **centroid coordinates** from the K-Means clustering model. Since the clustering was performed on **scaled data**, it is important to analyze the results both in the **scaled feature space** and after **inverse-scaling** back to the original feature values.

Why This Comparison Matters

- **K-Means centroids** are the **mean** of all data points assigned to each cluster in the feature space used for clustering.
- Our K-Means model was applied on **standardized (scaled)** data, so the **centroids are in scaled coordinates**.
- We compare:
  - The **scaled cluster means** with the **centroids in scaled space**.
  - The **original cluster means** (unscaled) with the **inverse-transformed centroids**.

Findings: Scaled Space Comparison (k=2)

Cluster	Feature	Scaled Mean	Centroid	Difference
0	CRIM	-0.3898	-0.3898	0.0000
	ZN	0.2624	0.2624	-0.0000
	INDUS	-0.6153	-0.6153	-0.0000
	CHAS	0.0029	0.0029	0.0000
	NOX	-0.5829	-0.5829	-0.0000
	RM	0.2449	0.2449	0.0000
	AGE	-0.4336	-0.4336	-0.0000
	DIS	0.4545	0.4545	-0.0000
	RAD	-0.5835	-0.5835	0.0000
	TAX	-0.6297	-0.6297	0.0000
	PTRATIO	-0.2947	-0.2947	0.0000
	B	0.3286	0.3286	0.0000
	LSTAT	-0.4535	-0.4535	-0.0000

Cluster	Feature	Scaled Mean	Centroid	Difference
	MEDV	0.3536	0.3536	0.0000

Cluster	Feature	Scaled Mean	Centroid	Difference
1	CRIM	0.7245	0.7245	-0.0000
	ZN	-0.4877	-0.4877	-0.0000
	INDUS	1.1437	1.1437	-0.0000
	CHAS	-0.0054	-0.0054	0.0000
	NOX	1.0835	1.0835	-0.0000
	RM	-0.4552	-0.4552	0.0000
	AGE	0.8059	0.8059	-0.0000
	DIS	-0.8448	-0.8448	0.0000
	RAD	1.0845	1.0845	-0.0000
	TAX	1.1705	1.1705	-0.0000
	PTRATIO	0.5477	0.5477	-0.0000
	B	-0.6108	-0.6108	0.0000
	LSTAT	0.8429	0.8429	-0.0000
	MEDV	-0.6573	-0.6573	-0.0000

- Differences are effectively **zero**, confirming **perfect match** between scaled cluster means and centroids.

Findings: Unscaled Space Comparison (k=2)

Cluster	Feature	Original Mean	Inverse-Scaled Centroid	Difference
0	CRIM	0.26	0.26	0.00
	ZN	17.48	17.48	-0.00
	INDUS	6.92	6.92	-0.00
	CHAS	0.07	0.07	0.00
	NOX	0.49	0.49	0.00
	RM	6.46	6.46	0.00
	AGE	56.38	56.38	0.00
	DIS	4.75	4.75	-0.00
	RAD	4.47	4.47	0.00
	TAX	302.21	302.21	0.00
	PTRATIO	17.82	17.82	0.00
	B	386.64	386.64	0.00
	LSTAT	9.42	9.42	0.00
	MEDV	25.78	25.78	-0.00

Cluster	Feature	Original Mean	Inverse-Scaled Centroid	Difference
1	CRIM	9.84	9.84	-0.00
	ZN	0.00	0.00	-0.00
	INDUS	18.98	18.98	-0.00
	CHAS	0.07	0.07	0.00
	NOX	0.68	0.68	0.00
	RM	5.97	5.97	0.00
	AGE	91.24	91.24	0.00
	DIS	2.02	2.02	0.00
	RAD	18.98	18.98	-0.00
	TAX	605.32	605.32	-0.00
	PTRATIO	19.64	19.64	-0.00
	B	300.97	300.97	0.00
	LSTAT	18.67	18.67	-0.00
	MEDV	16.49	16.49	0.00

- Differences are **negligible** and arise only due to **floating-point rounding** during scaling transformations.

2.5.6 Summary and Interpretation

- **Centroids = Cluster Means** in scaled space, confirming the mathematical definition of K-Means.
- After **inverse scaling**, centroids closely approximate the original feature means of clusters.
- Minor differences are expected due to **numerical precision** and do not affect interpretation.
- This confirms the **validity and accuracy** of our K-Means clustering results on the Boston dataset.

Conclusion:
The centroids are reliable representatives of cluster centers, allowing confident interpretation of cluster characteristics in both scaled and original feature spaces.

2.6 Citations:

- https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html
- https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
- https://www.datacamp.com/tutorial/k-means-clustering-python
- https://www.kaggle.com/code/nasimetemadi/clustring-of-customers

2.6.1 2.3 Problem 3

[17]:
```
# Step 1: Load the Wine Dataset
# Explanation: We load the Wine dataset from sklearn.datasets and convert it␣
↪into a Pandas DataFrame.
# The dataset includes 13 features and a target variable (class labels: 0, 1,␣
↪2) representing wine cultivars.
# We include the class labels in the DataFrame for later comparison with␣
↪cluster assignments.

import pandas as pd
from sklearn.datasets import load_wine

# Load the Wine dataset
wine = load_wine()

# Create DataFrame with feature names and data
df_wine = pd.DataFrame(data=wine.data, columns=wine.feature_names)

# Add the actual class labels as a column
df_wine['class'] = wine.target

# Display the first few rows to verify loading
print("Wine Dataset (First 5 Rows):")
print(df_wine.head())
```

```
Wine Dataset (First 5 Rows):
   alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  \
0    14.23        1.71  2.43               15.6      127.0           2.80
1    13.20        1.78  2.14               11.2      100.0           2.65
2    13.16        2.36  2.67               18.6      101.0           2.80
3    14.37        1.95  2.50               16.8      113.0           3.85
4    13.24        2.59  2.87               21.0      118.0           2.80

   flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  \
0        3.06                  0.28             2.29             5.64  1.04
1        2.76                  0.26             1.28             4.38  1.05
2        3.24                  0.30             2.81             5.68  1.03
3        3.49                  0.24             2.18             7.80  0.86
4        2.69                  0.39             1.82             4.32  1.04

   od280/od315_of_diluted_wines  proline  class
0                          3.92   1065.0      0
1                          3.40   1050.0      0
2                          3.17   1185.0      0
3                          3.45   1480.0      0
4                          2.93    735.0      0
```

```
[18]: # Step 2: Scale the Data
      # Explanation: We standardize all features (excluding the class label) to have␣
       ↪mean=0 and variance=1
      # using StandardScaler. This ensures fair distance calculations in K-Means␣
       ↪clustering.

      from sklearn.preprocessing import StandardScaler

      # Select only the feature columns (exclude 'class')
      features = wine.feature_names
      X = df_wine[features]

      # Apply standardization
      scaler = StandardScaler()
      X_scaled = scaler.fit_transform(X)

      # Convert scaled data back to a DataFrame for easier handling
      df_scaled = pd.DataFrame(X_scaled, columns=features)

      # Display the first few rows of scaled data
      print("\nScaled Wine Dataset (First 5 Rows):")
      print(df_scaled.head())
```

```
Scaled Wine Dataset (First 5 Rows):
    alcohol  malic_acid       ash  alcalinity_of_ash  magnesium  \
0  1.518613   -0.562250  0.232053          -1.169593   1.913905
1  0.246290   -0.499413 -0.827996          -2.490847   0.018145
2  0.196879    0.021231  1.109334          -0.268738   0.088358
3  1.691550   -0.346811  0.487926          -0.809251   0.930918
4  0.295700    0.227694  1.840403           0.451946   1.281985

   total_phenols  flavanoids  nonflavanoid_phenols  proanthocyanins  \
0       0.808997    1.034819             -0.659563         1.224884
1       0.568648    0.733629             -0.820719        -0.544721
2       0.808997    1.215533             -0.498407         2.135968
3       2.491446    1.466525             -0.981875         1.032155
4       0.808997    0.663351              0.226796         0.401404

   color_intensity       hue  od280/od315_of_diluted_wines   proline
0         0.251717  0.362177                      1.847920  1.013009
1        -0.293321  0.406051                      1.113449  0.965242
2         0.269020  0.318304                      0.788587  1.395148
3         1.186068 -0.427544                      1.184071  2.334574
4        -0.319276  0.362177                      0.449601 -0.037874
```

[21]:
```
# Step 3: Perform K-Means Clustering with k=3
# Explanation: We apply K-Means clustering on the scaled data with k=3, as␣
↪specified.
# The random_state is set for reproducibility, and n_init=10 ensures multiple␣
↪initializations for better results.

from sklearn.cluster import KMeans

# Initialize and fit K-Means with k=3
kmeans = KMeans(n_clusters=3, random_state=42, n_init=10)
kmeans.fit(X_scaled)

# Store cluster labels
cluster_labels = kmeans.labels_

# Show first 10 predicted cluster labels
print("Cluster labels for first 10 data points:", cluster_labels[:10])
```

```
Cluster labels for first 10 data points: [2 2 2 2 2 2 2 2 2 2]
```

[22]:
```
# Step 4: Assign Cluster Labels
# Explanation: We assign the predicted K-Means cluster labels to the original␣
↪DataFrame.
# This allows us to analyze how many data points fall into each cluster.

# Add cluster labels to the DataFrame
df_wine['cluster'] = cluster_labels

# Display the number of data points in each cluster
print("\nCluster Distribution (k=3):")
for cluster_id, count in df_wine['cluster'].value_counts().sort_index().items():
    print(f"  Cluster {cluster_id}: {count} samples")
```

```

Cluster Distribution (k=3):
  Cluster 0: 65 samples
  Cluster 1: 51 samples
  Cluster 2: 62 samples
```

[23]:
```
# Step 5: Calculate Homogeneity and Completeness Scores
# Explanation:
# Homogeneity measures whether each cluster contains only members of a single␣
↪class.
# Completeness measures whether all members of a given class are assigned to␣
↪the same cluster.
# Both metrics range from 0 to 1, where 1 is perfect agreement.
# We compare the true wine class labels with the K-Means cluster assignments to␣
↪evaluate clustering quality.
```

```
from sklearn.metrics import homogeneity_score, completeness_score

# Calculate Homogeneity and Completeness scores
homogeneity = homogeneity_score(df_wine['class'], cluster_labels)
completeness = completeness_score(df_wine['class'], cluster_labels)

# Display the evaluation results with clear formatting
print("\nClustering Evaluation Metrics (k=3):")
print(f"  Homogeneity Score: {homogeneity:.4f}  (Higher is better, max=1.0)")
print(f"  Completeness Score: {completeness:.4f}  (Higher is better, max=1.0)")
```

```
Clustering Evaluation Metrics (k=3):
  Homogeneity Score: 0.8788  (Higher is better, max=1.0)
  Completeness Score: 0.8730  (Higher is better, max=1.0)
```

[26]:
```
# Step 6: Interpret Homogeneity and Completeness Metrics
# Explanation:
# - Homogeneity measures if each cluster contains only points from a single␣
 ↪class.
# - Completeness measures if all points of a class are assigned to the same␣
 ↪cluster.
# Both metrics together provide complementary insights on cluster quality.

print("\nInterpretation of Clustering Metrics:\n")

print(f"Homogeneity Score: {homogeneity:.4f}")
print("  - Indicates the extent to which clusters contain only members of a␣
 ↪single class.")
print("  - A score of 1.0 means perfect homogeneity: each cluster corresponds␣
 ↪to exactly one class.")
print("  - Lower scores suggest that clusters include a mixture of different␣
 ↪classes.\n")

print(f"Completeness Score: {completeness:.4f}")
print("  - Reflects whether all members of a given class are assigned to the␣
 ↪same cluster.")
print("  - A score of 1.0 means perfect completeness: all points of a class are␣
 ↪in one cluster.")
print("  - Lower scores indicate that a class is spread across multiple␣
 ↪clusters.\n")

print("Summary:")
print("  - High homogeneity implies pure clusters without class mixing.")
print("  - High completeness implies minimal fragmentation of classes across␣
 ↪clusters.")
```

```
print("  - Together, these metrics assess how well clustering captures the true␣
↪underlying class structure.")
print("  - For our K-Means clustering (k=3), the scores suggest a strong␣
↪alignment with actual wine classes.")
```

```
Interpretation of Clustering Metrics:

Homogeneity Score: 0.8788
  - Indicates the extent to which clusters contain only members of a single
class.
  - A score of 1.0 means perfect homogeneity: each cluster corresponds to
exactly one class.
  - Lower scores suggest that clusters include a mixture of different classes.

Completeness Score: 0.8730
  - Reflects whether all members of a given class are assigned to the same
cluster.
  - A score of 1.0 means perfect completeness: all points of a class are in one
cluster.
  - Lower scores indicate that a class is spread across multiple clusters.

Summary:
  - High homogeneity implies pure clusters without class mixing.
  - High completeness implies minimal fragmentation of classes across clusters.
  - Together, these metrics assess how well clustering captures the true
underlying class structure.
  - For our K-Means clustering (k=3), the scores suggest a strong alignment with
actual wine classes.
```

2.6.2 Step 6: Detailed Interpretation of Clustering Metrics with Scores

We evaluate the performance of the K-Means clustering (with $\mathbf{k=3}$) using the **Homogeneity** and **Completeness** metrics, which compare the cluster assignments to the true class labels of the Wine dataset.

Homogeneity Score: 0.8788

- This metric measures the extent to which **each cluster contains only members of a single class**.
- A **perfect score of 1.0** means clusters are **completely pure** — every cluster corresponds to exactly one class.
- Our score of **0.8788** indicates **high cluster purity**, meaning that approximately **87.88% of the clustering structure reflects true class homogeneity**.
- The remaining **12.12%** difference suggests some **mixing of different classes within certain clusters**, but this is minimal.

Completeness Score: 0.8730

- This metric assesses whether **all samples from a given class are assigned to the same cluster**.
- A **score of 1.0** means each class is **entirely grouped within a single cluster**, without fragmentation.
- Our completeness score of **0.8730** implies that about **87.30% of the actual class membership is captured within individual clusters**.
- The residual **12.70%** indicates a small amount of **class fragmentation across clusters**, where some classes are split into multiple clusters.

2.6.3 Summary and Insights

- Both **Homogeneity (0.8788)** and **Completeness (0.8730)** scores are **close to 1**, which is a strong indicator that the clustering model is doing a very good job at capturing the true underlying class structure.
- **High homogeneity** confirms that clusters are largely **pure and contain samples from only one class**.
- **High completeness** means that **most of the members of a single class are grouped together** in the same cluster.
- The minor deviations from 1.0 (roughly 12%) reflect **some degree of overlap and class mixing**, which is expected in real-world data and unsupervised clustering.
- Therefore, the K-Means clustering with **3 clusters** aligns well with the **actual wine classes**, validating its effectiveness in this context.

2.7 Citations:

- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.homogeneity_score.html
- https://scikit-learn.org/stable/modules/generated/sklearn.metrics.completeness_score.html
- https://www.kaggle.com/code/abdallahwagih/k-means-clustering
- https://www.kaggle.com/code/digvijaysingh16/k-mean-clustering-for-wine-quality-data
- https://www.kaggle.com/code/thedatageek/clustering-eda-analysis-clearly-explained

```
[ ]:
```

```
[ ]:
```