**Лекція 4. Кластерізація**

* + - 1. **Поняття та призначення**
      2. **Кластерізація K-mean**
      3. **Алгоритм ієрархічної кластеризації**
      4. **Алгоритм DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**
      5. **Нейронні мережі для кластеризації**

1. **Поняття та призначення**

Кластеризація — це набір методів без учителя для групування даних за певними критеріями в так звані кластери, що дозволяє виявляти схожості та відмінності між об'єктами, а також спрощувати їх аналіз і візуалізацію. Через часткову схожість у постановці завдань з класифікацією кластеризацію ще називають unsupervised classification. (классифікація без підкріплення).

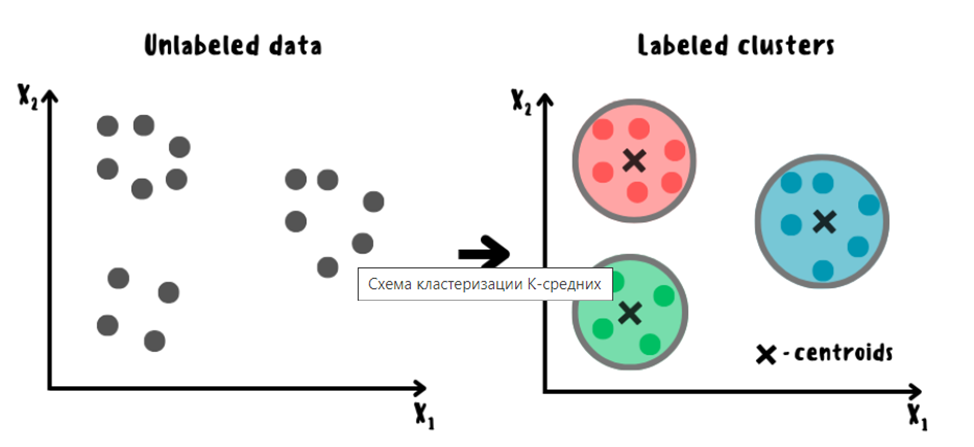
**Області застосування кластеризації та її різновиди**Кластеризація широко застосовується в машинному навчанні для вирішення різноманітного спектру завдань:

1. класифікація (визначення, до якого класу належить кожен об'єкт або виділення нових класів, які не були відомі заздалегідь);
2. сегментація ринку (поділ потенційних клієнтів на групи за їхніми характеристиками для розробки більш ефективних стратегій у маркетингу та продажах);
3. сегментація зображень (поділ зображення на сегменти або групи пікселів);
4. кластеризація геоданих (групування даних за їхнім географічним розташуванням, наприклад, поділ районів на безпечні та небезпечні, багаті й бідні тощо);
5. зниження розмірності (зменшення кількості ознак шляхом об'єднання схожих в один кластер).
6. **Кластеризація K-mean**

Сутність k-means (k-середніх) полягає в розбитті набору даних на k кластерів, де кожен кластер представлений своїм центроїдом (середнім значенням). Основна мета алгоритму — мінімізувати суму квадратів відстаней між кожним об'єктом і найближчим центроїдом.

Алгоритм k-means працює наступним чином:

1. Вибір кількості кластерів (k) — користувач задає кількість кластерів, на які потрібно поділити дані.
2. Ініціалізація центроїдів — випадковим чином вибираються початкові центроїди для кожного з кластерів.
3. Призначення об'єктів кластерам — кожен об'єкт даних призначається до найближчого центроїда (згідно з обраною метрикою, зазвичай це евклідова відстань).
4. Оновлення центроїдів — обчислюється новий центроїд для кожного кластера як середнє значення всіх об'єктів, що належать цьому кластеру.
5. Повторення кроків 3 і 4 — процес призначення об'єктів і оновлення центроїдів повторюється до тих пір, поки центроїди не перестануть змінюватися або зміни будуть мінімальними.
6. Зупинка — алгоритм зупиняється, коли більше немає значних змін у положенні центроїдів.



K-means простий у реалізації та застосуванні, але має певні обмеження: алгоритм чутливий до початкового розміщення центроїдів і може потрапити в локальний мінімум.

**Простий приклад к-mean.**

Розглянемо простий приклад обчислення k-means вручну для 4 точок та 2 кластерів (k=2).

**Вхідні дані (4 точки у 2D-просторі)**

P1=(1,1),P2=(2,1),P3=(4,3),P4=(5,4)

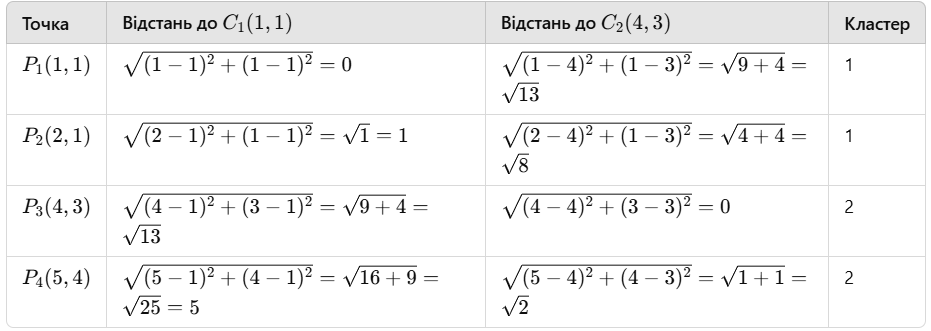
**Крок 1: Ініціалізація центрів кластерів**

Припустимо, що початкові центроїди вибрані випадково:

C1=(1,1),C2=(4,3)

**Крок 2: Призначення точок до кластерів**

Обчислюємо відстань (за Евклідовою метрикою) від кожної точки до центрів кластерів:

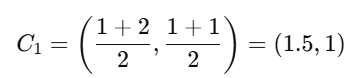


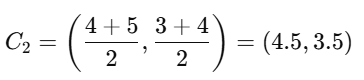
Нові групи:

* Кластер 1: P1(1,1),P2(2,1)
* Кластер 2: P3(4,3), P4(5,4)

**Крок 3: Оновлення центрів кластерів**

Обчислюємо нові центроїди:





### **Крок 4: Перевірка зупинки**

Якщо нові центроїди не змінилися або зміни малі — алгоритм завершується.  
Інакше — повторюємо процес призначення точок до кластерів та оновлення центроїдів.

**У нашому випадку центроїди змінилися, тож потрібно повторити крок 2 з новими центрами кластерів.**

Це триває, поки центри перестануть змінюватися.

**Приклад реалізації на Python.**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

# Створення ручного набору даних

# Кожен рядок - це точка в 2D-просторі (означення)

X = np.array([

[1, 2], [1, 4], [1, 0],

[4, 2], [4, 4], [4, 0],

[5, 1], [6, 1], [6, 2],

[5, 5], [6, 5], [5, 4]

])

# Візуалізація даних

plt.scatter(X[:, 0], X[:, 1], s=50)

plt.title("Випадкові дані")

plt.xlabel("Ознака 1")

plt.ylabel("Ознака 2")

plt.grid()

plt.show()

# Застосування алгоритму k-середніх

n\_clusters = 3

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=42)

kmeans.fit(X)

# Отримання центрів кластерів та передбачень

centers = kmeans.cluster\_centers\_

labels = kmeans.labels\_

# Візуалізація кластерів

plt.scatter(X[:, 0], X[:, 1], c=labels, s=50, cmap='viridis')

plt.scatter(centers[:, 0], centers[:, 1], c='red', s=200, alpha=0.75, marker='X', label='Центроїди')

plt.title("Результати кластеризації k-середніх")

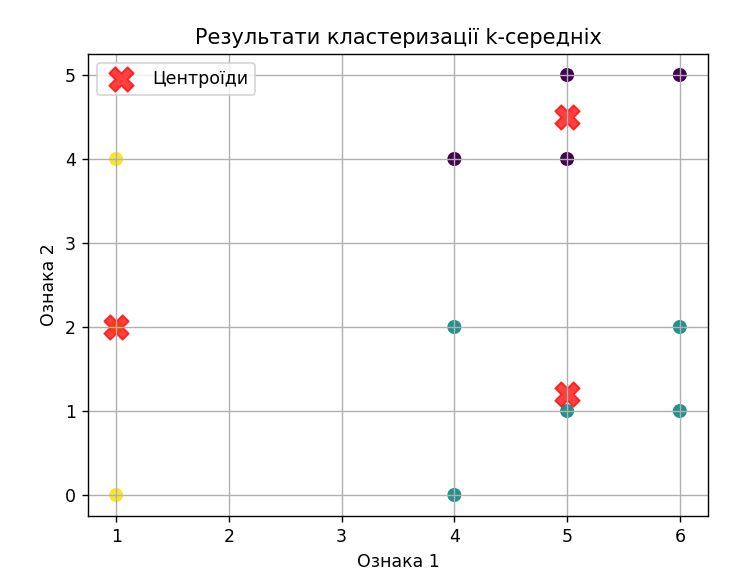
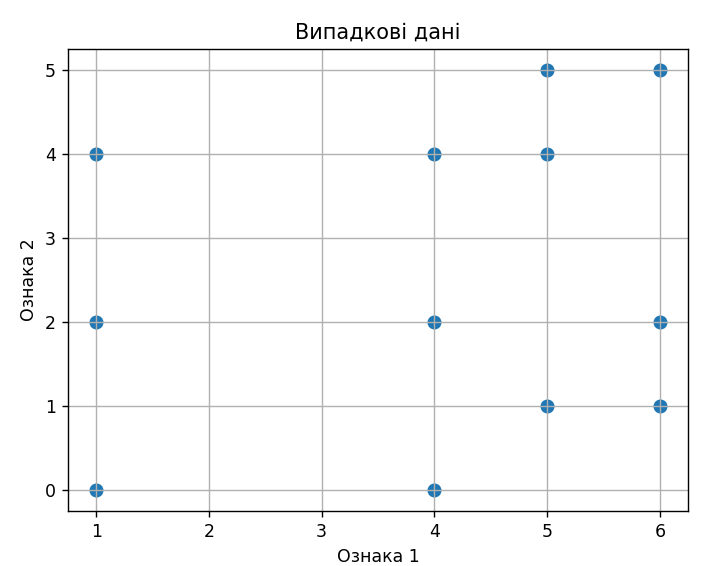
plt.xlabel("Ознака 1")

plt.ylabel("Ознака 2")

plt.legend()

plt.grid()

plt.show()



### Опис коду

1. **Створення набору даних**: Ми вручну визначили набір точок у 2D-просторі.
2. **Візуалізація даних**: Перший графік показує ці дані.
3. **Застосування k-середніх**: Ініціалізуємо модель KMeans з 3 кластерами та навчаємо її на ручному наборі даних.
4. **Отримання центрів кластерів**: Витягуємо координати центроїдів кластерів.
5. **Візуалізація результатів**: На другому графіку відображаємо дані з кольоровим маркуванням кластерів і центроїдами, позначеними червоними хрестиками.

**Модифікація k-mean++**

Метод **k-means++** — це вдосконалення стандартного алгоритму **k-means**, спрямоване на покращення ініціалізації центроїдів, що дозволяє уникнути проблем із вибором поганих початкових значень і зменшує ймовірність потрапляння в локальні мінімуми.

Основна відмінність полягає в розумному виборі початкових центроїдів, що дозволяє швидше збігатися до оптимального розв'язку.

Процес ініціалізації в k-means++:

1. **Вибір першого центроїда** — перший центроїд вибирається випадковим чином із набору даних.
2. **Вибір наступних центроїдів**:
   * Для кожної точки обчислюється відстань до найближчого вже обраного центроїда.
   * Кожна точка отримує вагу пропорційно квадрату цієї відстані.
   * Наступний центроїд вибирається випадковим чином, але з більшою ймовірністю для точок, які знаходяться на великій відстані від уже обраних центроїдів.
3. **Повторення** — крок 2 повторюється, поки не буде вибрано **k** центроїдів.

Після того як ініціалізація завершена, алгоритм працює так само, як і стандартний k-means: призначення точок до найближчих центроїдів і оновлення положень центроїдів на основі середніх значень точок у кластерах.

Переваги k-means++:

* **Швидша збіжність** — метод зменшує кількість ітерацій, необхідних для збіжності.
* **Краща якість кластерів** — центроїди краще розташовані, що дозволяє отримати більш рівномірний розподіл кластерів.
* **Зниження ризику локальних мінімумів** — правильна ініціалізація зменшує ймовірність того, що алгоритм потрапить у локальний мінімум.

**Приклад**

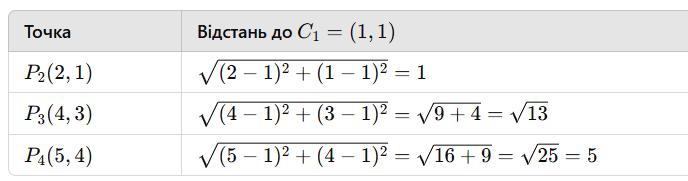
**Крок 1: Вибір першого центроїда (C₁) випадково**

Припустимо, що ми випадково обрали P1(1,1)) як перший центроїд:

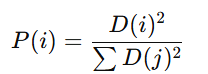
C1=(1,1)

**Крок 2: Вибір другого центроїда (C₂)**

Для кожної точки (яка ще не є центроїдом) обчислюємо її мінімальну відстань до вже обраного центру

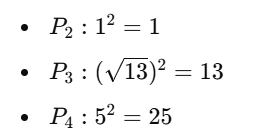
****

Обчислюємо ймовірність вибору кожної точки як нового центроїда:

****

де D(i)) — мінімальна відстань від точки iii до вже вибраного центроїда.

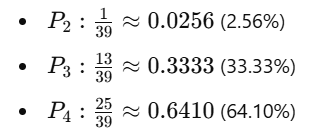
Обчислюємо квадрати відстаней:

****

Сума всіх квадратів відстаней:

****

Ймовірності вибору наступного центру:

****

Випадково вибираємо точку згідно з цими ймовірностями.

Припустимо, що випав

****

**Крок 3: Запуск звичайного k-means**

Тепер, коли центри **C1=(1,1)** **C2=(5,4)** обрані,  
ми можемо запустити алгоритм k-means, як у попередньому прикладі.

Він буде виконувати ітерації, поки центроїди не стабілізуються.

**В Python це реалізується за рахунок**

kmeans = KMeans(n\_clusters=n\_clusters, init='k-means++', random\_state=42)

**3.Алгоритм ієрархічної кластеризації**

Ієрархічна кластеризація будує дерево кластерів (дендрограму), що показує, як дані об'єднуються в кластери. Вона може бути агломеративною (об'єднує найближчі пункти) або дивергентною (розділяє пункти).

**Приклад**

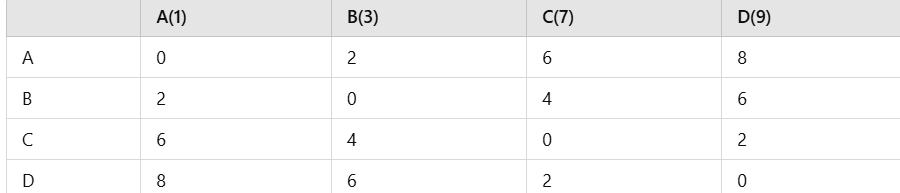
**Нехай у нас є 4 точки:**

A=1,B=3,C=7,D=9

(одновимірні значення для простоти).

Використаємо **метод найближчого сусіда** (**single linkage**) для об'єднання.

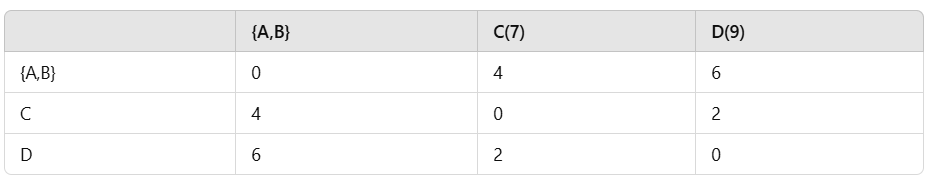
Знаходимо абсолютні різниці між точками (Евклідова відстань):



Об’єднуємо найближчі пари

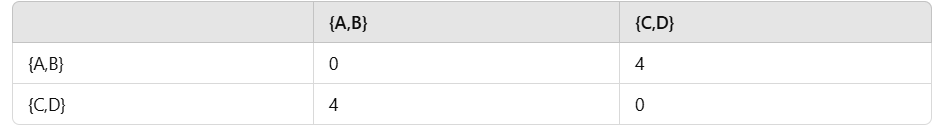
Найменша відстань 2 → точки A (1) і B (3) об’єднуються у кластер {A, B}.

Оновлюємо матрицю відстаней (беремо мінімальну відстань до інших):



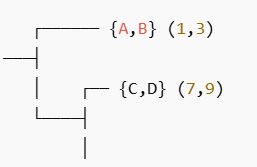
Об’єднуємо наступні найближчі точки

Найменша відстань 2 → точки C (7) і D (9) об’єднуються у кластер {C, D}.



Об’єднуємо два залишкові кластери

* Відстань між {A,B} і {C,D} — 4, тому їх об'єднуємо.
* В результаті всі точки зливаються у один кластер.



Приклад реалізації на Python

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_moons

from scipy.cluster.hierarchy import dendrogram, linkage

# Створення даних

X, \_ = make\_moons(n\_samples=30, noise=0.05, random\_state=42)

# Виконання ієрархічної кластеризації

linked = linkage(X, 'ward')

# Візуалізація дендрограми

plt.figure(figsize=(10, 5))

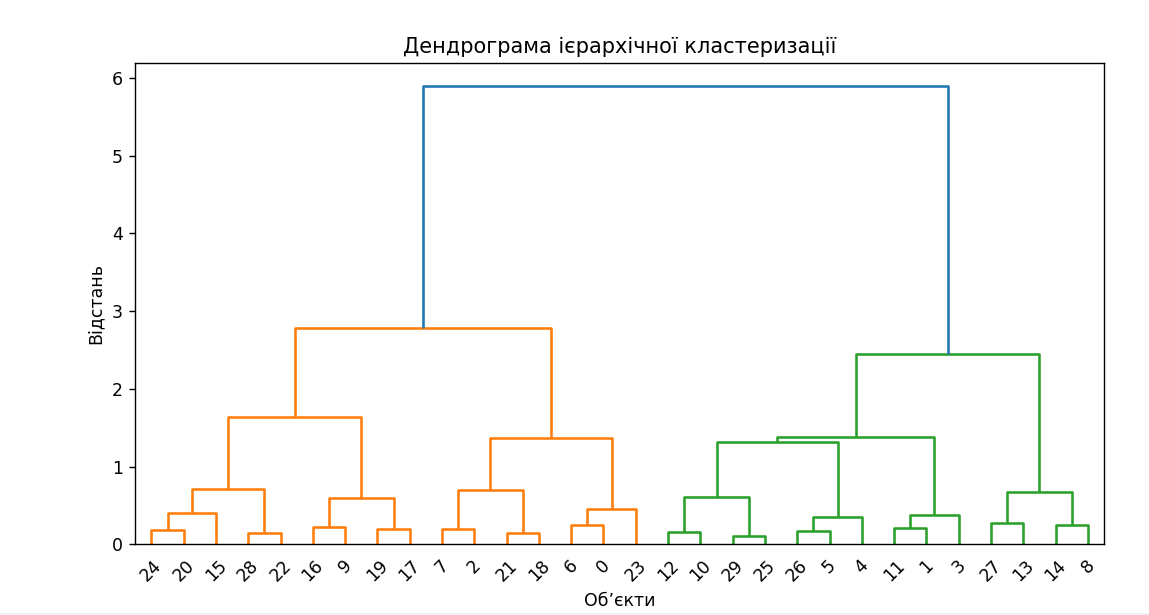
dendrogram(linked, orientation='top', distance\_sort='descending', show\_leaf\_counts=True)

plt.title('Дендрограма ієрархічної кластеризації')

plt.xlabel('Об’єкти')

plt.ylabel('Відстань')

plt.show()

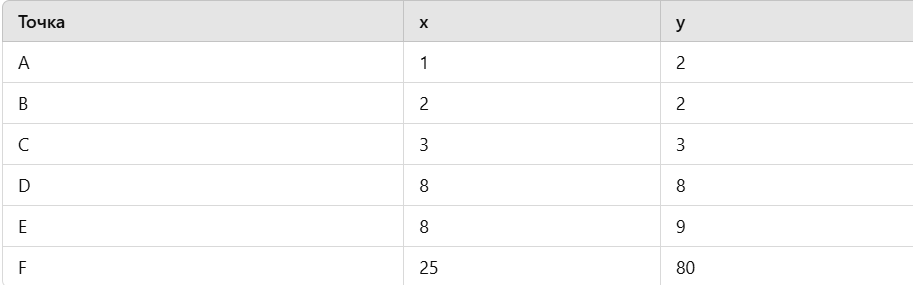


**4.Алгоритм DBSCAN (Density-Based Spatial Clustering of Applications with Noise)**

DBSCAN визначає кластери на основі густини даних. Він може виявляти кластери різної форми та стійкий до шуму.

**Приклад:**

Нехай у нас є 6 точок у 2D-просторі:



📌 Точки **(A, B, C)** близько одна до одної,  
📌 Точки **(D, E)** також поруч,  
📌 А **F** — далеко від усіх.

**Параметри DBSCAN**

* ε = 2.5 (радіус пошуку сусідів)
* min\_samples = 2 (мінімальна кількість точок для утворення кластеру)

**Алгоритм крок за кроком**

З**находимо сусідів для кожної точки (ε = 2.5)**

* **A (1,2)**: сусіди → {B}
* **B (2,2)**: сусіди → {A, C}
* **C (3,3)**: сусіди → {B}
* **D (8,8)**: сусіди → {E}
* **E (8,9)**: сусіди → {D}
* **F (25,80)**: сусідів немає

**Позначаємо ядрові та граничні точки**

* **B** (2 сусіди: A, C) → **Ядрова**
* **D** (2 сусіди: E) → **Ядрова**
* **A і C** → **Граничні (бо мають менше сусідів, ніж min\_samples)**
* **E** → **Гранична**
* **F** → **Шум (бо немає сусідів)** ❌

**Формуємо кластери**

* **{A, B, C} → Кластер 1**
* **{D, E} → Кластер 2**
* **F → Шумова точка (викид)** ❌

**🔹 Результат кластеризації**



import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import make\_moons

from sklearn.cluster import DBSCAN

# Створення даних

X, \_ = make\_moons(n\_samples=100, noise=0.1, random\_state=42)

# Виконання DBSCAN

dbscan = DBSCAN(eps=0.2, min\_samples=5)

labels = dbscan.fit\_predict(X)

# Візуалізація результатів

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis', s=50)

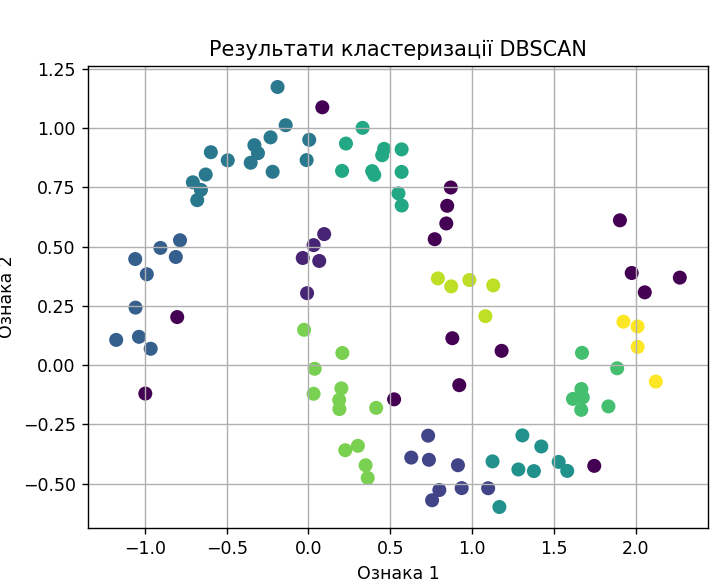
plt.title("Результати кластеризації DBSCAN")

plt.xlabel("Ознака 1")

plt.ylabel("Ознака 2")

plt.grid()

plt.show()



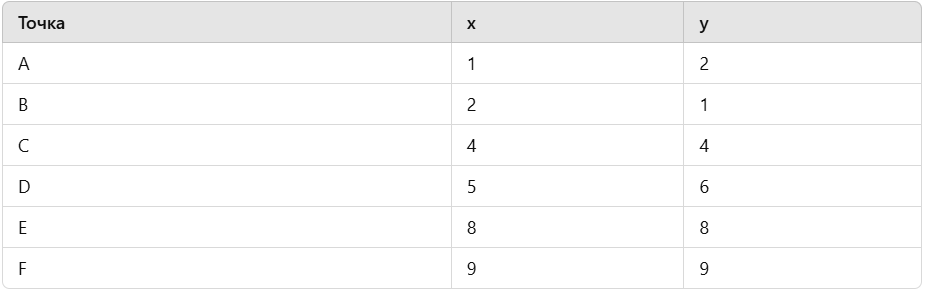
1. **Нейронні мережі для кластеризації**

### **Self-Organizing Maps (SOM)**

**Метод самоорганізуючих карт (Self Organizing Map, SOM)** — це тип штучної нейронної мережі, яка також натхненна біологічними моделями нейронних систем з 1970-х років. Він слідує підходу безкерованого навчання і тренує свою мережу через алгоритм конкурентного навчання. SOM використовується для кластеризації та картографування (або зменшення вимірності), що дозволяє відображати багатовимірні дані на нижчій вимірності, спрощуючи складні проблеми для легшої інтерпретації. SOM має два шари: вхідний шар і вихідний шар.

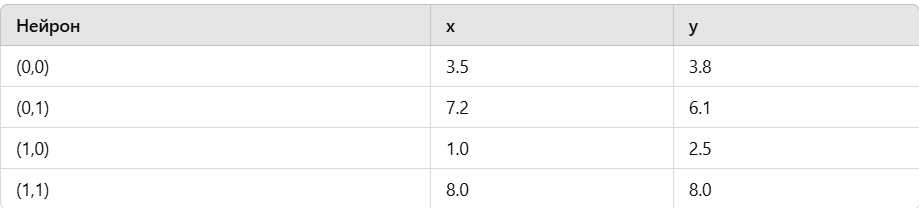
**Приклад**

У нас є **6 точок у 2D-просторі**:



Ми хочемо, щоб **SOM згрупував ці точки** в **2×2 карту нейронів** (4 нейрони), кожен з яких представлятиме групу схожих точок

Ми створюємо **2×2 карту** з 4 нейронів, кожен з яких має **випадкові вагові вектори (координати)**.



**Пошук найближчого нейрона (BMU - Best Matching Unit)**

Для кожної точки знаходимо **найближчий нейрон** за Евклідовою відстанню.

Наприклад, для точки **A(1,2)**:

* Відстань до нейрона (0,0) = sqrt((1-3.5)² + (2-3.8)²) ≈ 3.2
* Відстань до нейрона (1,0) = sqrt((1-1.0)² + (2-2.5)²) ≈ 0.5 ✅ (найближчий)
* Відстань до інших нейронів більша

Отже**, нейрон (1,0) є BMU для точки A.**

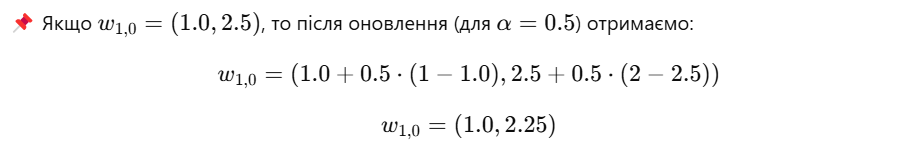
**Оновлення ваг нейрона та його сусідів**

Найближчий нейрон рухається ближче до вхідної точки, а його сусіди оновлюються слабше.

Формула оновлення ваг: 

де:

* Wі — вага нейрона
* α— швидкість навчання (наприклад, 0.5)
* x — координати точки



**Повторення кроків для всіх точок**

Кожна точка знаходить найближчий нейрон

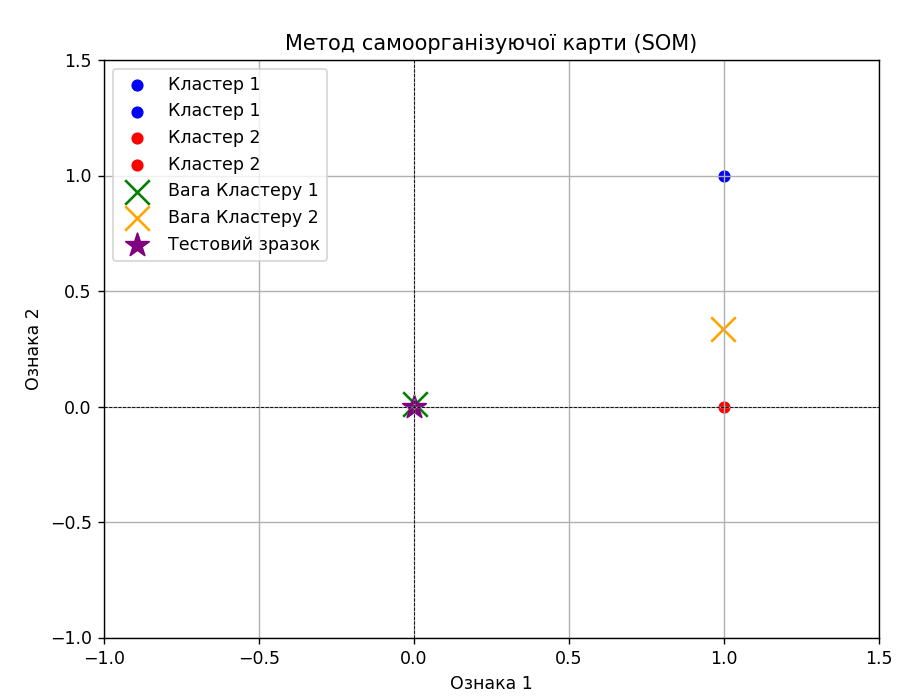
Ваги нейронів оновлюються

Радіус оновлення та швидкість навчання поступово зменшуються

Після кількох ітерацій нейрони стають представниками кластерів.

**Приклад реалізації на Python:**

import math  
import matplotlib.pyplot as plt  
  
# Приклади навчання (m, n)  
T = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]  
T = [[sample[0], sample[1]] for sample in T] # Використовуємо лише перші дві ознаки для візуалізації  
  
m, n = len(T), len(T[0])  
  
# Ініціалізація ваг (n, C)  
weights = [[0.2, 0.6], [0.8, 0.4]] # Використовуємо лише перші дві ознаки  
  
# Параметри навчання  
epochs = 3  
alpha = 0.5  
  
for \_ in range(epochs):  
 for j in range(m):  
 # Приклад навчання  
 sample = T[j]  
  
 # Обчислення виграшного вектора  
 D = [0, 0]  
 for i in range(len(sample)):  
 D[0] += math.pow((sample[i] - weights[0][i]), 2)  
 D[1] += math.pow((sample[i] - weights[1][i]), 2)  
  
 # Визначення виграшного індексу  
 J = 0 if D[0] < D[1] else 1  
  
 # Оновлення ваг виграшного вектора  
 for i in range(len(weights[J])):  
 weights[J][i] += alpha \* (sample[i] - weights[J][i])  
  
# Класифікація тестового зразка  
s = [0, 0]  
D = [0, 0]  
for i in range(len(s)):  
 D[0] += math.pow((s[i] - weights[0][i]), 2)  
 D[1] += math.pow((s[i] - weights[1][i]), 2)  
  
J = 0 if D[0] < D[1] else 1  
  
print("Тестовий зразок s належить до кластера: ", J)  
print("Треновані ваги: ", weights)  
  
# Візуалізація даних  
plt.figure(figsize=(8, 6))  
  
# Візуалізуємо навчальні приклади  
for i in range(m):  
 plt.scatter(T[i][0], T[i][1], c='blue' if i < 2 else 'red', label='Кластер 1' if i < 2 else 'Кластер 2')  
  
# Візуалізуємо ваги  
plt.scatter(weights[0][0], weights[0][1], c='green', marker='x', s=200, label='Вага Кластеру 1')  
plt.scatter(weights[1][0], weights[1][1], c='orange', marker='x', s=200, label='Вага Кластеру 2')  
  
# Візуалізуємо тестовий зразок  
plt.scatter(s[0], s[1], c='purple', marker='\*', s=200, label='Тестовий зразок')  
  
plt.title('Метод самоорганізуючої карти (SOM)')  
plt.xlabel('Ознака 1')  
plt.ylabel('Ознака 2')  
plt.legend()  
plt.grid()  
plt.xlim(-1, 1.5)  
plt.ylim(-1, 1.5)  
plt.axhline(0, color='black',linewidth=0.5, ls='--')  
plt.axvline(0, color='black',linewidth=0.5, ls='--')  
  
plt.show()



Якщо ми хочемо використовувати всі 4 ознаки, то візуалізація в двох вимірах не буде можлива без подальшої обробки даних, наприклад, за допомогою методу зменшення розмірності (таких як PCA або t-SNE) для відображення даних у 2D