# KLASTERYZACJA W MACHINE LEARNING

## Wprowadzenie do Klasteryzacji

Klasteryzacja to zadanie **uczenia nienadzorowanego** (unsupervised learning), w którym algorytm grupuje dane w klastry bez wcześniejszego poznania etykiet. W przeciwieństwie do klasyfikacji, nie mamy z góry określonych kategorii - algorytm sam odkrywa naturalne grupy w danych.

**Podstawowa różnica:**

* **Klasyfikacja** (supervised): Mamy etykiety → uczymy model → przewidujemy etykiety nowych danych
* **Klasteryzacja** (unsupervised): Nie mamy etykiet → szukamy naturalnych grup → odkrywamy strukturę danych

**Przykłady zastosowań klasteryzacji:**

* Segmentacja klientów w marketingu (grupy o podobnych zachowaniach zakupowych)
* Kompresja obrazów (grupowanie podobnych kolorów)
* Wykrywanie anomalii (punkty nie należące do żadnego klastra)
* Grupowanie dokumentów według tematyki
* Analiza danych biologicznych (grupowanie genów o podobnej ekspresji)
* Organizacja bibliotek zdjęć (grupowanie podobnych obrazów)

**Kluczowe pojęcia:**

* **Klaster** - grupa podobnych do siebie obiektów
* **Centroid** - punkt reprezentujący środek klastra
* **Odległość** - miara podobieństwa między punktami
* **Outlier** - punkt odstający, nie należący do żadnego klastra

## 1. K-MEANS

**Jak działa K-means?**

K-means to najpopularniejszy algorytm klasteryzacji. Dzieli dane na **k** klastrów, gdzie k jest określone przez użytkownika. Każdy klaster reprezentowany jest przez centroid - punkt będący średnią wszystkich punktów w klastrze.

**Algorytm krok po kroku:**

1. **Inicjalizacja**: Losowo wybierz k punktów jako początkowe centroidy
2. **Przypisanie**: Przypisz każdy punkt danych do najbliższego centroidu (powstają klastry)
3. **Aktualizacja**: Przelicz centroidy jako średnią arytmetyczną wszystkich punktów w każdym klastrze
4. **Iteracja**: Powtarzaj kroki 2-3 aż:
   * Centroidy przestaną się zmieniać (zbieżność)
   * Osiągnięto maksymalną liczbę iteracji
   * Zmiany są poniżej określonego progu

**Intuicja:** Wyobraź sobie, że masz 100 punktów na płaszczyźnie i chcesz je podzielić na 3 grupy. K-means najpierw losowo "rzuca" 3 znaczniki (centroidy), a następnie:

* Każdy punkt "przyciąga się" do najbliższego znacznika
* Znaczniki przesuwają się do środka swoich grup
* Proces powtarza się, aż wszystko się ustabilizuje

**Kluczowe parametry**

**n\_clusters (k)** - liczba klastrów:

* Najbardziej krytyczny parametr
* Musi być określony z góry
* Użyj metody łokcia lub współczynnika sylwetkowego do wyboru optymalnego k

**init** - metoda inicjalizacji centroidów:

* 'k-means++' (domyślnie, zalecane) - inteligentna inicjalizacja, która wybiera centroidy tak, aby były od siebie oddalone
* 'random' - losowy wybór k punktów jako centroidów
* K-means++ znacznie poprawia zbieżność i jakość wyników

**n\_init** - liczba uruchomień algorytmu:

* Domyślnie 10
* Algorytm uruchamiany jest n\_init razy z różnymi inicjalizacjami
* Wybierany jest najlepszy wynik (najniższa inercja)
* Zapobiega utknięciu w lokalnym minimum

**max\_iter** - maksymalna liczba iteracji:

* Domyślnie 300
* Limit iteracji dla jednego uruchomienia algorytmu

**random\_state** - ziarno losowości:

* Zapewnia powtarzalność wyników

**Metoda łokcia (Elbow Method)**

**Po co?** Pomaga wybrać optymalną liczbę klastrów k.

**Jak to działa:**

* Trenuj K-means dla różnych wartości k (np. 2-10)
* Dla każdego k oblicz **inercję** (suma kwadratów odległości punktów do ich centroidów)
* Narysuj wykres k vs inercja
* Szukaj "łokcia" - punktu, gdzie inercja przestaje szybko spadać

**Interpretacja:**

* Mała wartość k → duża inercja (dane słabo zgrupowane)
* Duża wartość k → mała inercja (każdy punkt może być swoim klastrem)
* **Łokieć** to kompromis: k wystarczająco duże, ale nie za duże

**Współczynnik sylwetkowy (Silhouette Score)**

**Po co?** Alternatywna metoda wyboru k, która ocenia jakość klasteryzacji.

**Jak to działa:** Dla każdego punktu oblicza się:

* **a** - średnia odległość do innych punktów w tym samym klastrze (spójność)
* **b** - średnia odległość do punktów w najbliższym innym klastrze (separacja)
* **Silhouette = (b - a) / max(a, b)**

**Wartości:**

* **+1**: Punkt idealnie dopasowany do swojego klastra
* **0**: Punkt na granicy między klastrami
* **-1**: Punkt prawdopodobnie przypisany do złego klastra

**Średnia dla wszystkich punktów:**

* Bliżej 1 = lepsze klastry (zwarte i dobrze oddzielone)
* Bliżej 0 = klastry się nakładają
* Wartości ujemne = źle dobrana liczba klastrów

**Davies-Bouldin Index**

**Po co?** Kolejna metryka jakości klasteryzacji.

**Jak to działa:**

* Mierzy stosunek: (rozproszenie w klastrach) / (odległość między klastrami)
* Niższe wartości = lepsze klastry

**Interpretacja:**

* **Wartość 0**: Perfekcyjna klasteryzacja
* **Im niższa, tym lepiej**: Klastry zwarte i oddalone od siebie
* Wartości typowo w zakresie 0.5-2.5

**Implementacja w Pythonie (Metoda łokcia i optymalizacja)**

from sklearn.cluster import KMeans

from sklearn.datasets import make\_blobs

from sklearn.metrics import silhouette\_score, davies\_bouldin\_score

import matplotlib.pyplot as plt

# Generowanie danych przykładowych

X\_blobs, y\_true = make\_blobs(n\_samples=300, centers=4,

cluster\_std=0.60, random\_state=42)

# Metoda łokcia (Elbow Method) - wybór optymalnej liczby klastrów

inertias = []

silhouette\_scores = []

K\_range = range(2, 11)

for k in K\_range:

kmeans = KMeans(n\_clusters=k, random\_state=42, n\_init=10)

kmeans.fit(X\_blobs)

inertias.append(kmeans.inertia\_)

silhouette\_scores.append(silhouette\_score(X\_blobs, kmeans.labels\_))

# Wizualizacja metody łokcia

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

ax1.plot(K\_range, inertias, 'bo-')

ax1.set\_xlabel('Liczba klastrów (k)')

ax1.set\_ylabel('Inercja (suma kwadratów odległości)')

ax1.set\_title('Metoda łokcia')

ax1.grid(True)

ax2.plot(K\_range, silhouette\_scores, 'ro-')

ax2.set\_xlabel('Liczba klastrów (k)')

ax2.set\_ylabel('Współczynnik sylwetkowy')

ax2.set\_title('Współczynnik sylwetkowy dla różnych k')

ax2.grid(True)

plt.tight\_layout()

plt.show()

# K-means z optymalną liczbą klastrów

optimal\_k = 4

kmeans = KMeans(

n\_clusters=optimal\_k,

init='k-means++', # inteligentna inicjalizacja

n\_init=10,

max\_iter=300,

random\_state=42

)

y\_kmeans = kmeans.fit\_predict(X\_blobs)

# Wizualizacja wyników

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.scatter(X\_blobs[:, 0], X\_blobs[:, 1], c=y\_true, cmap='viridis', alpha=0.6)

plt.title('Prawdziwe etykiety')

plt.xlabel('Cecha 1')

plt.ylabel('Cecha 2')

plt.subplot(1, 2, 2)

plt.scatter(X\_blobs[:, 0], X\_blobs[:, 1], c=y\_kmeans, cmap='viridis', alpha=0.6)

plt.scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1],

s=300, c='red', marker='X', edgecolors='black', linewidths=2, label='Centroidy')

plt.title('Klasteryzacja K-means')

plt.xlabel('Cecha 1')

plt.ylabel('Cecha 2')

plt.legend()

plt.tight\_layout()

plt.show()

# Metryki ewaluacji

print(f"Współczynnik sylwetkowy: {silhouette\_score(X\_blobs, y\_kmeans):.4f}")

print(f"Davies-Bouldin Index: {davies\_bouldin\_score(X\_blobs, y\_kmeans):.4f}")

print(f"Inercja: {kmeans.inertia\_:.2f}")

**Co robi ten kod:**

1. Tworzy syntetyczne dane z 4 klastrami (gaussowskimi "bąbelkami")
2. Testuje K-means dla k od 2 do 10
3. Rysuje wykresy inercji i sylwetki - pomagają wybrać optymalne k
4. Trenuje model z optymalnym k=4
5. Wizualizuje prawdziwe etykiety vs wyniki K-means
6. Wyświetla metryki jakości

**Zalety i wady K-means**

**Zalety:**

* Bardzo szybki i skalowalny - działa dobrze z dużymi zbiorami danych
* Prosty w implementacji i zrozumieniu
* Gwarantowana zbieżność (zatrzyma się w skończonej liczbie iteracji)
* Działa dobrze z "kulowymi" klastrami (okrągłymi, o podobnej wielkości)
* Mało pamięci - przechowuje tylko centroidy

**Wady:**

* **Musi znać k z góry** - nie odkrywa liczby klastrów automatycznie
* Wrażliwy na inicjalizację - może utknąć w lokalnym minimum (dlatego n\_init > 1)
* Zakłada klastry kulowe o podobnej wielkości
* **Nie radzi sobie z:**
  + Klastrami o nieregularnych kształtach (księżyce, pierścienie)
  + Klastrami o różnej gęstości
  + Outlierami - każdy punkt musi być w jakimś klastrze
* Wrażliwy na skalę cech - wymaga standaryzacji

**Kiedy używać K-means?**

**Użyj K-means gdy:**

* Wiesz lub możesz oszacować liczbę klastrów
* Klastry mają kulowy kształt i podobną wielkość
* Potrzebujesz szybkiego algorytmu dla dużych danych
* Dane są w miarę równomiernie rozłożone

**Unikaj K-means gdy:**

* Klastry mają nieregularne kształty (użyj DBSCAN)
* Klastry mają bardzo różną gęstość
* Masz dużo outlierów
* Nie wiesz ile jest klastrów

## 2. DBSCAN (Density-Based Spatial Clustering)

**Jak działa DBSCAN?**

DBSCAN to algorytm oparty na **gęstości** - identyfikuje klastry jako obszary o wysokiej gęstości punktów, oddzielone obszarami o niskiej gęstości. W przeciwieństwie do K-means, DBSCAN:

* Nie wymaga określania liczby klastrów z góry
* Automatycznie wykrywa outliers
* Radzi sobie z klastrami o nieregularnych kształtach

**Rodzaje punktów w DBSCAN:**

1. **Core point** (punkt rdzeniowy):
   * Ma co najmniej min\_samples punktów w promieniu eps
   * Stanowi "rdzeń" klastra
2. **Border point** (punkt brzegowy):
   * Ma mniej niż min\_samples sąsiadów
   * Ale jest w zasięgu eps od core point
   * Należy do klastra, ale na jego obrzeżu
3. **Noise point** (outlier):
   * Nie jest core point
   * Nie jest w zasięgu żadnego core point
   * Oznaczany etykietą -1

**Algorytm krok po kroku:**

1. Wybierz losowy punkt, który nie był jeszcze odwiedzony
2. Znajdź wszystkie punkty w promieniu eps od niego
3. Jeśli jest ich co najmniej min\_samples:
   * Utwórz nowy klaster
   * Rekurencyjnie dodawaj wszystkie osiągalne punkty do klastra
4. Jeśli nie ma wystarczająco sąsiadów:
   * Oznacz jako noise (może być później przypisany jako border point)
5. Powtarzaj dla wszystkich nieodwiedzonych punktów

**Kluczowe parametry**

**eps (epsilon)** - promień sąsiedztwa:

* Maksymalna odległość między dwoma punktami, aby uznać je za sąsiadów
* **Najbardziej krytyczny parametr**
* Za małe eps → wszystko to noise
* Za duże eps → wszystko to jeden klaster
* Metoda wyboru: wykres k-distance (opisany poniżej)

**min\_samples** - minimalna liczba sąsiadów:

* Minimalna liczba punktów w promieniu eps, aby utworzyć core point
* Typowo: 2 × liczba wymiarów (dla 2D → min\_samples=4 lub 5)
* Większe wartości → bardziej konserwatywne klastry, więcej noise
* Mniejsze wartości → więcej małych klastrów

**metric** - metryka odległości:

* 'euclidean' (domyślnie) - standardowa odległość w linii prostej
* 'manhattan' - suma różnic bezwzględnych
* 'cosine' - dla danych tekstowych
* Inna niestandardowa metryka

**Wybór parametru eps - metoda k-distance**

**Problem:** Jak wybrać odpowiednie eps?

**Rozwiązanie - wykres k-distance:**

1. Dla każdego punktu znajdź odległość do k-tego najbliższego sąsiada (gdzie k = min\_samples)
2. Posortuj te odległości rosnąco
3. Narysuj wykres
4. Szukaj "łokcia" - punktu, gdzie krzywa gwałtownie rośnie

**Interpretacja:**

* **Przed łokciem**: Punkty gęsto upakowane (wewnątrz klastrów)
* **Po łokciu**: Punkty rzadkie (między klastrami lub noise)
* **Punkt łokcia** = optymalne eps

**Implementacja w Pythonie (DBSCAN podstawowy)**

from sklearn.cluster import DBSCAN

from sklearn.datasets import make\_moons

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import NearestNeighbors

import numpy as np

import matplotlib.pyplot as plt

# Generowanie danych o nieregularnym kształcie

X\_moons, y\_moons = make\_moons(n\_samples=300, noise=0.05, random\_state=42)

# Standaryzacja (ważna dla DBSCAN!)

scaler = StandardScaler()

X\_moons\_scaled = scaler.fit\_transform(X\_moons)

# Znalezienie optymalnego eps - metoda k-distance

neighbors = NearestNeighbors(n\_neighbors=5)

neighbors.fit(X\_moons\_scaled)

distances, indices = neighbors.kneighbors(X\_moons\_scaled)

# Sortowanie odległości

distances = np.sort(distances[:, -1], axis=0)

plt.figure(figsize=(10, 6))

plt.plot(distances)

plt.xlabel('Punkty posortowane według odległości')

plt.ylabel('5-ta najbliższa odległość')

plt.title('Wykres k-distance dla wyboru eps')

plt.grid(True)

plt.show()

# DBSCAN z wybranymi parametrami

dbscan = DBSCAN(

eps=0.3, # promień sąsiedztwa

min\_samples=5, # minimalna liczba punktów

metric='euclidean'

)

y\_dbscan = dbscan.fit\_predict(X\_moons\_scaled)

# Liczba klastrów i outlierów

n\_clusters = len(set(y\_dbscan)) - (1 if -1 in y\_dbscan else 0)

n\_noise = list(y\_dbscan).count(-1)

print(f"Liczba klastrów: {n\_clusters}")

print(f"Liczba outlierów: {n\_noise}")

# Wizualizacja wyników

fig, axes = plt.subplots(1, 3, figsize=(18, 5))

# Prawdziwe etykiety

axes[0].scatter(X\_moons[:, 0], X\_moons[:, 1], c=y\_moons, cmap='viridis', alpha=0.6)

axes[0].set\_title('Prawdziwe etykiety')

axes[0].set\_xlabel('Cecha 1')

axes[0].set\_ylabel('Cecha 2')

# K-means (dla porównania)

from sklearn.cluster import KMeans

kmeans\_moons = KMeans(n\_clusters=2, random\_state=42)

y\_kmeans\_moons = kmeans\_moons.fit\_predict(X\_moons)

axes[1].scatter(X\_moons[:, 0], X\_moons[:, 1], c=y\_kmeans\_moons, cmap='viridis', alpha=0.6)

axes[1].scatter(kmeans\_moons.cluster\_centers\_[:, 0], kmeans\_moons.cluster\_centers\_[:, 1],

s=300, c='red', marker='X', edgecolors='black', linewidths=2)

axes[1].set\_title('K-means (nie radzi sobie)')

axes[1].set\_xlabel('Cecha 1')

axes[1].set\_ylabel('Cecha 2')

# DBSCAN

axes[2].scatter(X\_moons[:, 0], X\_moons[:, 1], c=y\_dbscan, cmap='viridis', alpha=0.6)

# Oznacz outliery tylko jeśli istnieją

outliers = X\_moons[y\_dbscan == -1]

if len(outliers) > 0:

axes[2].scatter(outliers[:, 0], outliers[:, 1], c='red', marker='x',

s=100, linewidths=2, label='Outliers')

axes[2].legend() # Legenda tylko gdy są outliers

axes[2].set\_title(f'DBSCAN (radzi sobie dobrze) - {n\_noise} outliers')

axes[2].set\_xlabel('Cecha 1')

axes[2].set\_ylabel('Cecha 2')

plt.tight\_layout()

plt.show()

**Co robi ten kod:**

1. Tworzy dane w kształcie dwóch "księżyców" - niemożliwe dla K-means!
2. Standaryzuje dane (DBSCAN wymaga porównywalnych skal)
3. Używa metody k-distance do znalezienia optymalnego eps
4. Trenuje DBSCAN i identyfikuje klastry + outliery
5. Porównuje wyniki: prawda vs K-means vs DBSCAN
6. Pokazuje przewagę DBSCAN nad K-means dla nieregularnych kształtów

**Problem z różną gęstością**

from sklearn.datasets import make\_blobs

from sklearn.cluster import DBSCAN

import matplotlib.pyplot as plt

# Dane z klastrami o różnej gęstości

X\_varied, y\_varied = make\_blobs(n\_samples=[100, 50, 150],

centers=[[0, 0], [3, 3], [6, 1]],

cluster\_std=[0.5, 1.5, 0.3],

random\_state=42)

# DBSCAN na danych o różnej gęstości

dbscan\_varied = DBSCAN(eps=0.5, min\_samples=5)

y\_dbscan\_varied = dbscan\_varied.fit\_predict(X\_varied)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.scatter(X\_varied[:, 0], X\_varied[:, 1], c=y\_varied, cmap='viridis', alpha=0.6)

plt.title('Prawdziwe klastry (różna gęstość)')

plt.xlabel('Cecha 1')

plt.ylabel('Cecha 2')

plt.subplot(1, 2, 2)

plt.scatter(X\_varied[:, 0], X\_varied[:, 1], c=y\_dbscan\_varied, cmap='viridis', alpha=0.6)

outliers\_varied = X\_varied[y\_dbscan\_varied == -1]

if len(outliers\_varied) > 0:

plt.scatter(outliers\_varied[:, 0], outliers\_varied[:, 1], c='red',

marker='x', s=100, linewidths=2, label='Outliers')

plt.title('DBSCAN (problemy z różną gęstością)')

plt.xlabel('Cecha 1')

plt.ylabel('Cecha 2')

plt.legend()

plt.tight\_layout()

plt.show()

**Co pokazuje ten kod:**

* Tworzy 3 klastry o różnym odchyleniu standardowym (różna gęstość)
* DBSCAN ma problem: jedno eps nie pasuje do wszystkich
* Zbyt małe eps → gęste klastry się rozpadają
* Zbyt duże eps → rzadkie klastry łączą się
* To główna wada DBSCAN!

**Zalety i wady DBSCAN**

**Zalety:**

* **Nie wymaga określenia liczby klastrów k** - odkrywa automatycznie
* Radzi sobie z klastrami o dowolnym kształcie (księżyce, spirale, pierścienie)
* **Automatycznie wykrywa outliers** - punkty odstające oznaczone jako -1
* Odporne na wartości początkowe - deterministyczny wynik
* Tylko 2 parametry do dostrojenia (eps, min\_samples)

**Wady:**

* **Wrażliwe na parametry eps i min\_samples** - trudny dobór
* Problemy z klastrami o **różnej gęstości**
* Nie radzi sobie z danymi wysokowymiarowymi (curse of dimensionality)
* Wolniejsze niż K-means dla bardzo dużych zbiorów
* Brak centroidów - nie można łatwo przewidzieć klastra dla nowego punktu
* Wymaga standaryzacji danych

**Kiedy używać DBSCAN?**

**Użyj DBSCAN gdy:**

* Nie wiesz ile jest klastrów
* Klastry mają nieregularne kształty
* Chcesz automatycznie wykrywać outliers
* Klastry mają podobną gęstość
* Masz dane przestrzenne (geograficzne)

**Unikaj DBSCAN gdy:**

* Klastry mają bardzo różną gęstość (użyj HDBSCAN)
* Dane wysokowymiarowe (>20 wymiarów)
* Potrzebujesz bardzo szybkiego algorytmu dla milionów punktów
* Chcesz przewidywać klaster dla nowych punktów

## 3. PORÓWNANIE K-MEANS VS DBSCAN

**Kompleksowe porównanie na różnych danych**

from sklearn.datasets import make\_circles, make\_blobs, make\_moons

from sklearn.cluster import KMeans, DBSCAN

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import silhouette\_score

import matplotlib.pyplot as plt

datasets = [

make\_blobs(n\_samples=300, centers=3, cluster\_std=0.6, random\_state=42),

make\_moons(n\_samples=300, noise=0.05, random\_state=42),

make\_circles(n\_samples=300, noise=0.05, factor=0.5, random\_state=42)

]

dataset\_names = ['Kulowe klastry', 'Klastry księżycowe', 'Klastry okrągłe']

fig, axes = plt.subplots(3, 3, figsize=(15, 15))

for i, (X, y) in enumerate(datasets):

# Standaryzacja

X\_scaled = StandardScaler().fit\_transform(X)

# Oryginalne dane

axes[i, 0].scatter(X[:, 0], X[:, 1], c=y, cmap='viridis', alpha=0.6)

axes[i, 0].set\_title(f'{dataset\_names[i]} - Oryginał')

axes[i, 0].set\_ylabel('Cecha 2')

# K-means

kmeans = KMeans(n\_clusters=2, random\_state=42)

y\_kmeans = kmeans.fit\_predict(X)

axes[i, 1].scatter(X[:, 0], X[:, 1], c=y\_kmeans, cmap='viridis', alpha=0.6)

axes[i, 1].scatter(kmeans.cluster\_centers\_[:, 0], kmeans.cluster\_centers\_[:, 1],

s=200, c='red', marker='X', edgecolors='black', linewidths=2)

axes[i, 1].set\_title(f'K-means (Silhouette: {silhouette\_score(X, y\_kmeans):.3f})')

# DBSCAN

dbscan = DBSCAN(eps=0.3, min\_samples=5)

y\_dbscan = dbscan.fit\_predict(X\_scaled)

axes[i, 2].scatter(X[:, 0], X[:, 1], c=y\_dbscan, cmap='viridis', alpha=0.6)

outliers = X[y\_dbscan == -1]

if len(outliers) > 0:

axes[i, 2].scatter(outliers[:, 0], outliers[:, 1], c='red',

marker='x', s=100, linewidths=2)

# Oblicz silhouette tylko dla punktów nie będących outlierami

mask = y\_dbscan != -1

if mask.sum() > 0 and len(set(y\_dbscan[mask])) > 1:

sil\_score = silhouette\_score(X[mask], y\_dbscan[mask])

else:

sil\_score = 0

axes[i, 2].set\_title(f'DBSCAN (Silhouette: {sil\_score:.3f})')

for ax in axes[-1]:

ax.set\_xlabel('Cecha 1')

plt.tight\_layout()

plt.show()

**Co pokazuje ten kod:**

* Testuje oba algorytmy na 3 różnych typach danych
* **Kulowe klastry**: K-means wygrywa (prostsze i szybsze)
* **Księżyce**: DBSCAN zdecydowanie lepszy
* **Okręgi koncentryczne**: DBSCAN radzi sobie, K-means nie
* Pokazuje współczynnik sylwetkowy dla porównania jakości

**Tabela porównawcza**

| **Cecha** | **K-means** | **DBSCAN** |
| --- | --- | --- |
| **Liczba klastrów** | Trzeba znać z góry | Wykrywa automatycznie |
| **Kształt klastrów** | Kulowy | Dowolny |
| **Outliers** | Zawsze przypisuje do klastra | Automatycznie wykrywa |
| **Szybkość** | Bardzo szybki | Wolniejszy |
| **Parametry** | k, init, n\_init | eps, min\_samples |
| **Standaryzacja** | Zalecana | Konieczna |
| **Różna gęstość** | Nie radzi sobie | Problemy |
| **Wysokie wymiary** | Średnio | Słabo |
| **Przewidywanie nowych punktów** | Łatwe | Trudne |
| **Interpretowalność** | Wysoka (centroidy) | Średnia |

**Kiedy co wybrać?**

**Wybierz K-means gdy:**

* Masz kulowe klastry o podobnej wielkości
* Wiesz lub możesz oszacować liczbę klastrów
* Potrzebujesz szybkości i skalowalności
* Chcesz prostą interpretację (centroidy)
* Przykład: segmentacja klientów, kompresja kolorów

**Wybierz DBSCAN gdy:**

* Nie wiesz ile jest klastrów
* Klastry mają nieregularne kształty
* Masz outliery do wykrycia
* Dane przestrzenne lub geograficzne
* Przykład: analiza anomalii, grupowanie lokalizacji GPS

## ZADANIA PRAKTYCZNE

**Zadanie 1: Eksperyment z liczbą klastrów w K-means**

Użyj datasetu Iris i:

* Przetestuj K-means dla k=2,3,4,5
* Narysuj metodę łokcia i współczynnik sylwetkowy
* Które k jest optymalne? Dlaczego?
* Porównaj z prawdziwą liczbą gatunków (3)

**Zadanie 2: Wpływ inicjalizacji w K-means**

Uruchom K-means z init='random' i n\_init=1 kilka razy:

* Czy dostajesz za każdym razem takie same wyniki?
* Porównaj z init='k-means++' i n\_init=10
* Dlaczego k-means++ jest lepszy?

**Zadanie 3: Dobór parametrów DBSCAN**

Na danych make\_moons:

* Przetestuj różne wartości eps (0.1, 0.3, 0.5, 1.0)
* Przetestuj różne min\_samples (3, 5, 10)
* Jak parametry wpływają na liczbę klastrów i outlierów?
* Znajdź optymalne parametry używając wykresu k-distance

**Zadanie 4: Porównanie na własnych danych**

Wygeneruj dane używając make\_circles:

X, y = make\_circles(n\_samples=500, noise=0.05, factor=0.5, random\_state=42)

* Zastosuj K-means i DBSCAN
* Który algorytm działa lepiej? Dlaczego?
* Co się stanie jak zmienisz parametr factor?

**Zadanie 5: Analiza outlierów**

Użyj DBSCAN na danych z ręcznie dodanymi outlierami:

X, \_ = make\_blobs(n\_samples=300, centers=3, random\_state=42)

# Dodaj outliery

outliers = np.random.uniform(low=-10, high=10, size=(20, 2))

X = np.vstack([X, outliers])

* Ile outlierów wykrył DBSCAN?
* Czy są to rzeczywiście punkty odstające?
* Co się stanie gdy zwiększysz eps?

**Zadanie 6: Klasteryzacja rzeczywistych danych**

Załaduj dataset Wine lub Breast Cancer:

* Usuń etykiety i zastosuj klasteryzację
* Porównaj wyniki z prawdziwymi etykietami
* Która metoda lepiej odkrywa naturalne grupy?

**METRYKI EWALUACJI KLASTERYZACJI**

Ponieważ klasteryzacja jest uczeniem nienadzorowanym, nie mamy "prawdziwych" etykiet do porównania. Używamy wewnętrznych metryk jakości:

**1. Inercja (Within-Cluster Sum of Squares)**

* Suma kwadratów odległości punktów do ich centroidów
* Niższa = lepiej
* Używana w metodzie łokcia
* Zawsze maleje wraz ze wzrostem k

**2. Współczynnik sylwetkowy (Silhouette Score)**

* Zakres: [-1, 1]
* Mierzy jak dobrze punkt pasuje do swojego klastra vs innych
* Wyższy = lepiej
* Idealny dla wyboru k

**3. Davies-Bouldin Index**

* Zakres: [0, ∞]
* Niższy = lepiej
* Stosunek rozproszenia wewnątrz do separacji między klastrami
* 0 = perfekcyjna klasteryzacja

**4. Calinski-Harabasz Index (Variance Ratio)**

* Zakres: [0, ∞]
* Wyższy = lepiej
* Stosunek rozproszenia między klastrami do wewnątrz klastrów
* Szybszy do obliczenia niż sylwetka

**Jak używać:**

from sklearn.metrics import (

silhouette\_score,

davies\_bouldin\_score,

calinski\_harabasz\_score

)

print(f"Silhouette: {silhouette\_score(X, labels):.3f}")

print(f"Davies-Bouldin: {davies\_bouldin\_score(X, labels):.3f}")

print(f"Calinski-Harabasz: {calinski\_harabasz\_score(X, labels):.3f}")

## PODSUMOWANIE

**K-means:**

* Prosty, szybki, popularny
* Wymaga znać k
* Klastry kulowe
* Świetny punkt startowy

**DBSCAN:**

* Automatycznie wykrywa k
* Dowolne kształty klastrów
* Wykrywa outliers
* Wrażliwy na parametry

**Złota zasada:** Zawsze zaczynaj od wizualizacji danych! Kształt i struktura danych podpowie, który algorytm wybrać. Testuj oba i porównuj wyniki używając metryk i wizualizacji.

**Praktyczny workflow:**

1. Zwizualizuj dane (scatter plot)
2. Standaryzuj cechy
3. Jeśli klastry kulowe → K-means (metoda łokcia dla k)
4. Jeśli nieregularne → DBSCAN (k-distance dla eps)
5. Oblicz metryki jakości
6. Zwaliduj wyniki wizualnie
7. Jeśli niezadowolony, dostosuj parametry i powtórz