POLITECHNIKA RZESZOWSKA
im. IGNACEGO ŁUKASIEWICZA

WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

DBSCAN

Projektowanie systemów i sieci komputerowych

Mateusz Śliwa
Piotr Świder

Spis treści

1. Wprowadzenie.

Algorytm DBSCAN (ang. Density-based spatial clustering of applications with noise) jest algorytmem klasteryzacji danych opartym na gęstości. Został on wymyślony w 1996 roku przez Martina Estera. Klastry utworzone za pomocą algorytmu charakteryzują się dużym zagęszczeniem punktów w stosunku do otoczenia. Algorytm umożliwia tworzenie klastrów o dowolnej wielkości oraz kształcie.

2. Charakterystyka.

2.1. Parametry.

Aby poprawnie opisać algorytm DBCAN niezbędne będzie użycie dwóch parametrów.

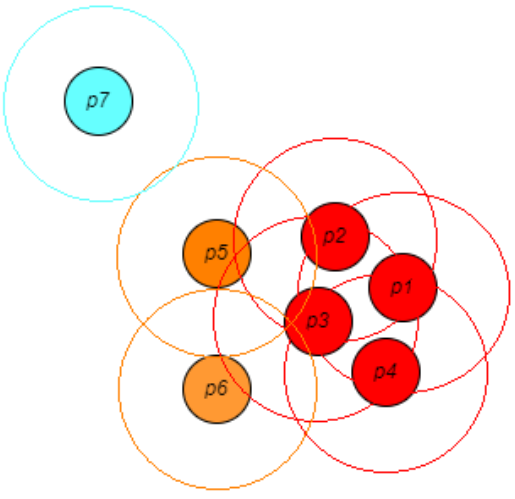
- ε *(eps, epsilon)* będzie określać maksymalny promień sąsiedztwa między punktami.
- *minPts* oznaczający minimalną liczbę punktów (obiektów) w klastrze.

2.2. Główne założenia.

Zakładając pewny zbiór X punktów (obiektów), oraz znając dwa parametry z poprzedniego podpunktu, można utworzyć odpowiedni klaster. Należy jednak wcześniej wprowadzić pewne oznaczenia oraz nazewnictwo punktów, mianowicie: punkt centralny, punkt osiągalny (szczególnie punkt bezpośrednio osiągalny), punkt szumu (*noise*).

- Punkt $p \in X$ jest punktem centralnym, gdy w promieniu ε od punktu p znajduje się minPts punktów (wliczając w to punkt p.)
- Punkt $q \in X$ jest punktem bezpośrednio osiągalnym gdy znajduje się on w odległości ε od punktu p.
- Punkt $q \in X$ jest punktem osiągalnym gdy istnieje ścieżka prowadząca między punktami p oraz q.
- Punkty szumu, są to wszystkie pozostałe nieosiągalne punkty.

Zbiór wszystkich punktów osiągalnych oraz punktu p tworzą odpowiedni klaster, tak jak na poniższym przykładowym schemacie:



Schemat tworzenia klastrów w algorytmie DBSCAN

Zakładając, że *p1* jest punktem centralnym oraz *minPts*=3:

- *p2, p3, p4* są punktami bezpośrednio osiągalnymi
- *p5, p6* są punktami osiągalnymi
- *p7* jest punktem szumu

Utworzony klaster: *p1, p2, p3, p4, p5, p6*

2.3. Sposób działania.

Sposób działania algorytmu DBSCAN można przedstawić w następujących krokach:

1. Wybierz bądź wylosuj punk *a* ze zbioru X.
2. Wyszukaj punkty ze zbioru X, których odległość od punktu *a* jest mniejsza bądź równa od *ε*.
3. Jeśli ilość znalezionych punktów jest większa bądź równa od *minPts,* ustal punt *a* jako punkt centralny *p*, następnie:
   I. Utwórz klaster znaleziony w punkcie *(2),* składający się z punktu centralnego *p* oraz punktów bezpośrednio osiągalnych.
   II. Wyszukuj oraz dodaj kolejne punkty osiągalne do klastra, które posiadają ścieżkę prowadzącą do punktu *p*.
4. Wybierz kolejny punkt *a*, pomijając punkty będące już częścią klastrów.
5. Jeśli żaden punkt nie spełnia już odpowiednich założeń, przerwij działanie algorytmu. Punkty, które nie znalazły się w żadnym klastrze potraktuj jako punkty szumu.

2.4. Złożoność.

Wpływ na wydajność działania algorytmu ma głównie odpowiednie ustalenie parametrów, dla zadanego zbioru danych. Ogólna średnia złożoność algorytmu DBSCAN wynosi O(*n log n*), a w najgorszym wypadku (przykładowo gdy wszystkie punkty znajdują się od siebie w większej odległości niż *ε,* co powoduje że każdy punkt zostaje sprawdzony) wynosi ona O(n^2).

3. Implementacja.

3.1. Kod algorytmu.

Poniższy kod algorytmu DBSCAN został napisany w Python, na podstawie przedstawionego pseudokodu.

```
import math

# Funkcja sprawdzająca czy odległość pomiędzy punktem orientacyjnym Q, a między punktem
z zestawu danych jest mniejsza lub równa podanemu epsilon.

def RangeQuery(DB, distFunc, Q, eps):
    Point_Neighbors = []
    for point in DB:
        if distFunc(Q, point) <= eps:
            Point_Neighbors.append(point)

    return Point_Neighbors

# Funkcja obliczająca odległość pomiędzy dwoma punktami.

def distFunc(point1, point2):
```

```
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)

# Algorytm klastrujący podane punkty.

def DBSCAN_Algorithm(DB, distFunc, eps, minPts):
    Cluster_Counter = 0
    Labels = [0] * len(DB)

    for i in range(len(DB)):
        if Labels[i] != 0:
            continue

        Neighbors = RangeQuery(DB, distFunc, DB[i], eps)


        if len(Neighbors) < minPts:
            Labels[i] = -1  # -1 - szum
            continue

        Cluster_Counter += 1
        Labels[i] = Cluster_Counter
        Seeds = Neighbors

        j = 0
        while j < len(Seeds):
            Point = Seeds[j]

            if Labels[DB.index(Point)] == -1:
                Labels[DB.index(Point)] = Cluster_Counter

            elif Labels[DB.index(Point)] == 0:
                Labels[DB.index(Point)] = Cluster_Counter
                Point_Neighbors = RangeQuery(DB, distFunc, Point, eps)

                if len(Point_Neighbors) >= minPts:
                    Seeds.extend(Point_Neighbors)

            j += 1

    return Labels

labels = DBSCAN_Algorithm(DB, distFunc, eps, minPts)

print("Etykiety klastrów:", labels)
```

3.2. Opis kodu.

Algorytm został podzielony na 3 części (funkcje), aby cały skrypt był bardziej czytelny. Pierwszą funkcją jest RangeQuery przyjmująca poszczególne argumenty: DB (bazę punktów), distFunc (druga funkcja), Q (punkt, od którego obliczamy odległość) oraz eps (epsilon) odległość, którą musi spełnić dana para punktów, aby mogły należeć do tego samego klastra.

Drugą funkcją jest distFunc, która oblicza odległość pomiędzy parą punktów. Korzystamy w niej ze wzoru:

$$|XY| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

DBSCAN_Algoritm to implementacja całego algorytmu wykorzystująca dwie poprzednie funkcję – RangeQuery oraz distFunc (RangeQuery wykorzystuje distFunc jako argument).

- Zaczynamy od stworzenia licznika klastrów, który ustawiamy na 0 oraz od stworzenia etykiet, które będą oznaczeniem konkretnych klastrów (przykładowy pokaz działania w następnej sekcji).
- Następnie przechodzimy do przejścia przez punkty z bazy DB – jeśli punkt ma etykietę równą 0 to znaczy, że nie został jeszcze przypisany do żadnego klastra, a jeśli ma różną od 0, to znaczy, że go pomijamy, ponieważ otrzymał już swój klaster i idziemy dalej.
- Wykorzystujemy funkcję RangeQuery, aby obliczyć i wskazać sąsiednie punkty.
- Stawiamy warunek: jeśli dany punkt nie ma wystarczającej liczby sąsiadów oznaczamy go jako punkt szumu (-1).
- Tworzymy nowy klaster i dodajemy do licznika jedynkę.
- Następnie, dopóki wszyscy sąsiedzi przypisani jako Seeds nie zostaną przetworzeni pętla będzie rozszerzać aktualny klaster.
- Jednym z końcowych kroków jest sprawdzenie etykiet – jeśli punkt wcześniej był szumem, to teraz jest przypisywany do klastra – analogicznie, jeśli punkt był = 0 teraz zaczyna należeć do klastra.
- Ostatnim krokiem jest ponowne znalezienie i dodanie nowych sąsiadów.
- Funkcja na końcu zwraca wszystkie etykiety podanych punktów (tzn. kolejno punkty do jakich klastrów należą).

3.3. Przykładowe działanie.

W połączeniu wszystkich omówionych wszystkich funkcji, skrypt dla podanej bazy DB, oraz zadanych parametrów *eps, minPts* działa w sposób następujący:

Przypadek 1.
Poniższy przykład przedstawia działanie algorytmu dla specjalnie wybranych danych testowych oraz parametrów *eps = 3, minPts = 2*, aby odpowiednio pokazać działanie algorytmu.
Baza punktów:

```
DB = [
    [1, 1], [2, 2], [2, 3], [8, 8],
    [8, 9], [25, 80], [24, 81], [23, 79],
    [50, 50], [51, 51], [52, 52]
]
```

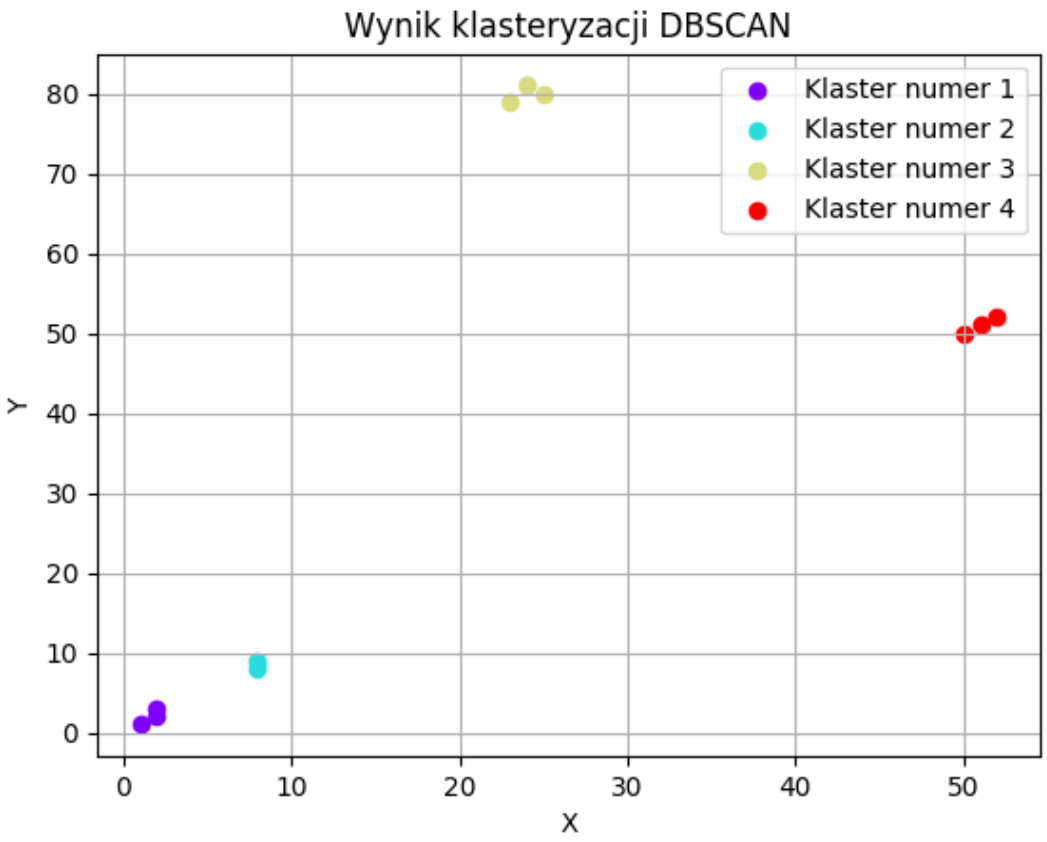
Dane dla przypadku 1

Wynik skryptu:

```
Etykiety klastrów: [1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4]
```

Wynik działania algorytmu dla przypadku 1

Przykładowa wizualizacja wyniku skryptu:

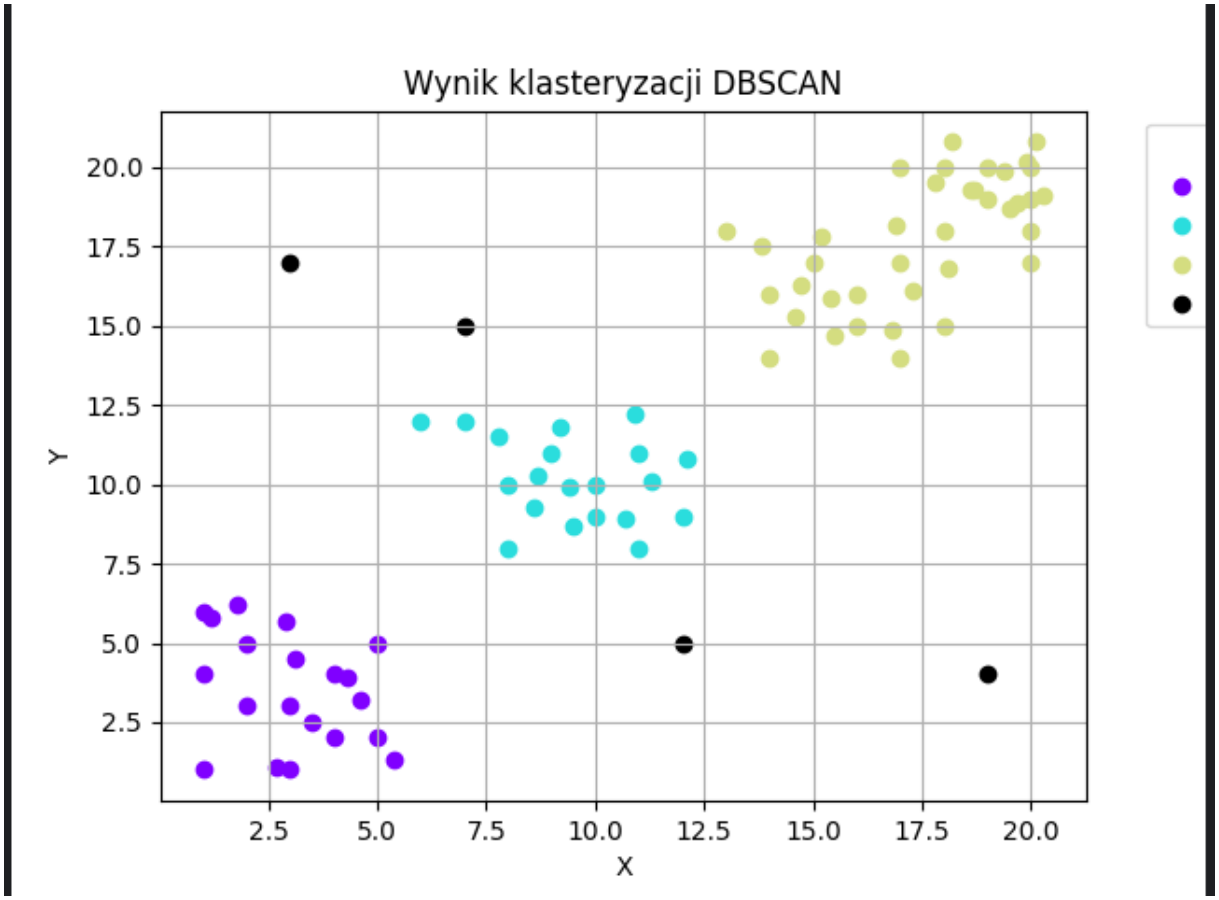


Wykres wyników dla przypadku 1.

Przypadek 2.
W przypadku, gdy wprowadzone dane będą na mniejszym przedziale (np. 1-20) wizualizacja będzie wyglądała bardziej czytelnie. Na poniższym wykresie zostały wprowadzone dane w większej ilości oraz na mniejszym przedziale.

Gdzie parametry wynoszą *minPts = 2* oraz *eps = 2*, czarny kolor oznacza punkty szumu, a pozostałe kolory to poszczególne klastry.

Wizualizacja wyniku skryptu:



Wykres wyników dla przypadku 2.

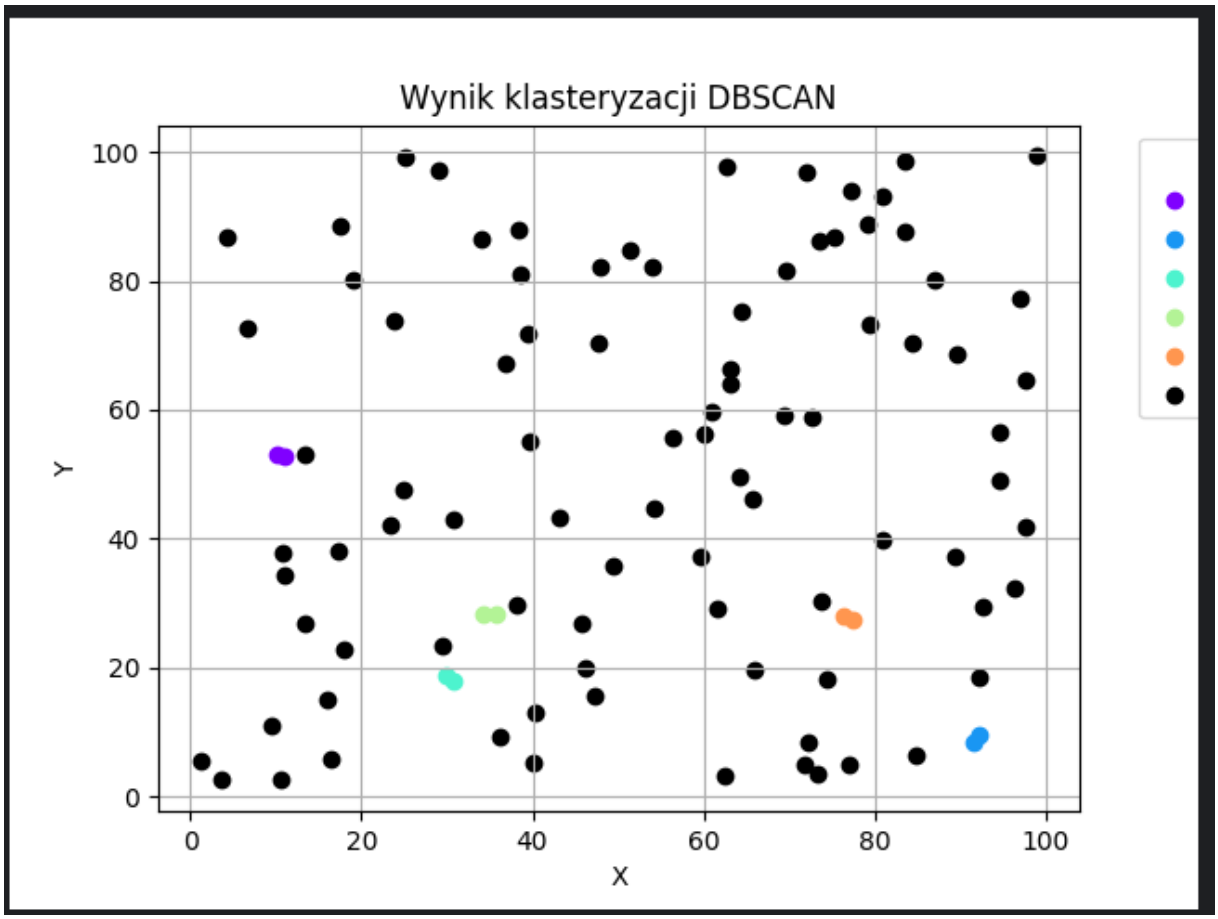
Przypadek 3.
Dla większej ilości bardziej losowych danych będzie to wyglądać całowicie inaczej. Przykładowo, gdy *eps* będzie równe 1.8 oraz weźmie się pod uwagę 100 punktów, gdzie ich wartość może być równa od 1-100, będzie można zauważyć ogromną różnicę.

Generowanie losowych danych:

```
DB = []
for x in range(1,100):
    DB.append([random.uniform( a: 1.00,  b: 100.00), random.uniform( a: 1.00, b: 100.00)])
```

Funkcja generująca losowe dane

Wizualizacja wyniku skryptu:



Wykres wyników dla przypadku 3.

W aktualnie omawianym przypadku można zaobserwować jeden z minusów działania algorytmu DBSCAN. Przy wysoko rozbieżnych danych działanie algorytmu jest zaburzone. Wpływ na to mają również parametry, których dobór jest kluczowy, aby algorytm DBSCAN przynosił pożądane wyniki i działał poprawnie.

4. Podsumowanie.

DBSCAN jest niezwykle użytecznym algorytmem, który swoje zastosowania odnalazł w wielu dziedzinach, jak choćby analiza danych czy też uczenie maszynowe. Jednak jak każde narzędzie posiada on swoje wady i zalety.

4.1. Zalety.

DBCSAN nie potrzebuje wcześniejszego określenia wielkości klastra ani ilości klastrów co umożliwia dynamiczne ich tworzenie. Dodatkowo może on służyć do wykrywa szumu (*noise*). Dzięki temu, że wymaga jedynie dwóch parametrów jest on stosunkowo prosty w zrozumieniu oraz implementacji.

4.2. Wady.

Algorytm DBSCAN nie jest tak optymalny w przypadku wysoce zróżnicowanych danych, w wypadku których ciężko może być ustalić i wybrać odpowiednie wartości parametrów. Użyteczność i jego zastosowanie zależy w dużej mierze od wybrania parametru ε. Nie umożliwia on określania liczby tworzonych klastrów.

4.3. Zastosowania.

Z racji na specyficzne cechy algorytmu DBSCAN posiada on zróżnicowane zastosowania.

- Dzięki możliwości wykrywania punktów szumu algorytmu DBSCAN używa się do wykrywania anomalii w danych.
- DBSCAN można zastosować w marketingu do identyfikacji klastrów klientów o podobnych preferencjach.
- W wypadku uczenia maszynowego, algorytmu DBSCAN używa się do wizualizacji oraz znajdywania pewnych wzorców oraz struktur.
- Algorytm DBSCAN jest używany w analizie przestrzennej do identyfikacji podobnych regionów lub regionów o zbliżonych właściwościach.
- W analizie danych, umożliwia on wnioskowanie na podstawie gęstości ułożenia danych.

4.4 Wnioski

Użyteczność algorytmu DBCSAN wynika głównie, przez jego unikalne zdolności oraz co bardzo istotne, w prostocie jego implementacji. Zrozumienie sposobu działania algorytmu jest wyjątkowo proste co przekłada się na zwiększenie jego popularności. Wykrywanie szumu, będące jednym z najważniejszych aspektów algorytmu DBSCAN, jest niezwykle użyteczne w analizie danych, co w połączeniu z łatwym do zrozumienia algorytmem umożliwia powszechne jego zastosowanie do wykrywania anomalii, błędnych danych oraz ich usunięcia. Warto mieć na uwadze, że algorytm DBSCAN nie jest też narzędziem bez wad, oraz posiada on pewną specyfikację użyteczności oraz zastosowania.

Bibliografia

Clustering Like a Pro: A Beginner's Guide to DBSCAN. (2024). Retrieved from Medium: https://medium.com/@sachinsoni600517/clustering-like-a-pro-a-beginners-guide-to-dbscan-6c8274c362c4

DBSCAN. (2024). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/DBSCAN

DBSCAN. (2024). Retrieved from Wikipedia: https://pl.wikipedia.org/wiki/DBSCAN