WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

Metody eksploracji danych w odkrywaniu wiedzy

Dokumentacja końcowa

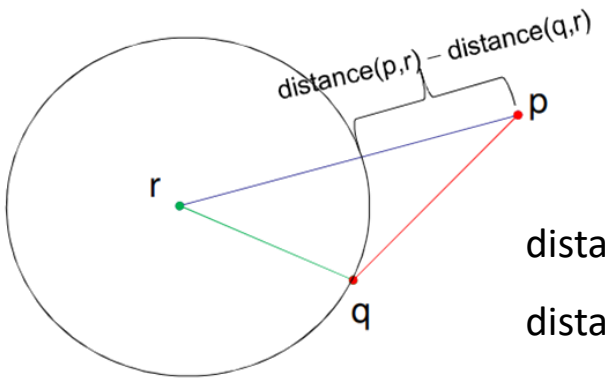
Paweł Martyniuk – nr albumu: 300220

Wprowadzenie i definicja problemu

W ramach projektu miałem za zadanie zaimplementować algorytm grupowania danych oparty na sąsiedztwie z wykorzystaniem nierówności trójkąta opisany w artykule pt.: „A Neighborhood-Based Clustering by Means of the Triangle Inequality” [1]. Grupowanie, w kontekście analizy danych, stanowi istotny proces polegający na identyfikowaniu istniejących wzorców i struktur wewnątrz zbioru danych. Celem tego działania jest zorganizowanie danych w logiczne i znaczące grupy, co umożliwia lepsze zrozumienie relacji pomiędzy nimi. Wynikiem procesu grupowania jest przypisanie etykiet do obiektów, co pozwala na łatwiejszą interpretację oraz analizę informacji zawartych w zbiorze.

Charakterystyka proponowanego rozwiązania

Na podstawie artykułu chciałbym przedstawić ideę rozwiązania na podstawie której zaimplementowałem algorytm. Istotnym elementem algorytmu przedstawionego w artykule jest własność nierówności trójkąta, która ma zastosowanie w obliczaniu odległości między dwoma różnymi punktami. Dzięki temu możemy pesymistycznie oszacować odległość bez kosztownego obliczania dokładnej wartości tak jak zostało to pokazane na rysunku 1.



$$distance(p,q) + distance(q,r) \geq distance(p,r)$$

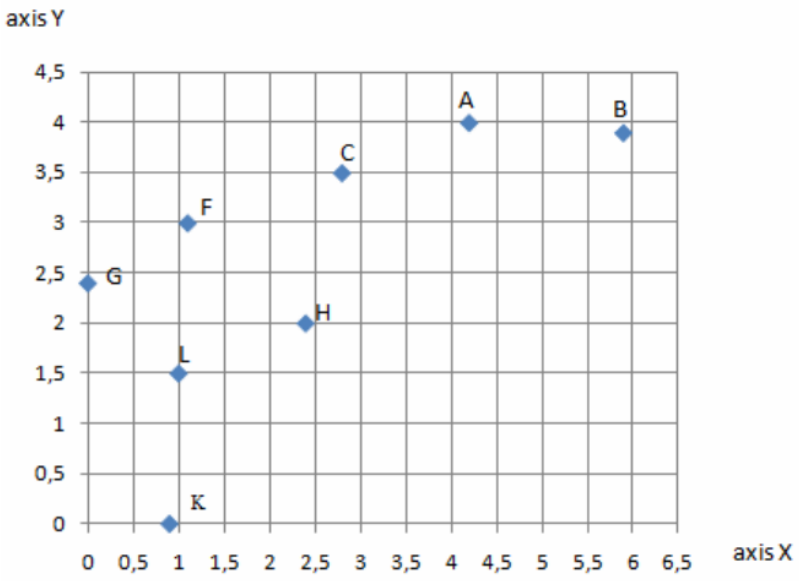
$$distance(p,q) \geq distance(p,r) - distance(q,r)$$

Rysunek 1 Własność nierówności trójkąta

Chciałbym wprowadzić następujące oznaczenia z których będę korzystać przy omawianiu algorytmu:

- k^+-NN – sąsiedztwo danego punktu - zbiór wszystkich punktów dla których dla których odległość od tego punktu nie przekracza odległości do najdalszego sąsiada
- Rk^+-NN – odwrotne sąsiedztwo danego punktu - zbiór wszystkich punktów dla których dany punkt jest sąsiadem
- $NDF(p) = \frac{|Rk^+\text{-}NN(p)|}{k^+\text{-}NN(p)}$ - współczynnik gęstości

Omówię algorytm dla przykładowych danych na rysunku 2.



Rysunek 2 Przykładowe dane do grupowania

Rozpocznę wyznaczanie sąsiedztwa od punktu F natomiast wartość k = 3 (minimalna liczba sąsiadów). Na początku należy obliczyć odległość każdego punktu do punktu referencyjnego. Dla tych danych dobrym punktem będzie punkt (0,0). Następnie obliczone wartości należy zapisać do pamięci i posortować niemalejąco względem tej odległości tak jak na rysunku 3.

q∈ D	X	Y	distance(q,r)
K	0,9	0,0	0,9
L	1,0	1,5	1,8
G	0,0	2,4	2,4
H	2,4	2,0	3,1
F	**1,1**	**3,0**	**3,2**
C	2,8	3,5	4,5
A	4,2	4,0	5,8
B	5,9	3,9	7,1

Rysunek 3 Posortowane punkty względem obliczonego dystansu

Jako że k = 3 to wybieram z posegregowanej tabeli 3 punkty dla których odległość do punktu referencyjnego jest najbardziej zbliżona do tej dla punktu F, która się równa 3,2. W związku z tym wybieram punkty H, G, C i obliczam dla nich rzeczywistą odległość do punktu F i są to następujące wartości: 1,64, 1,25 oraz 1,77. Jako epsilon wybieram wartość maksymalną z tych trzech czyli 1,77. Więc mamy gwarancję że dla epsilona = 1,77 mamy 3 sąsiadów dla tego punktu. Następnie musimy sprawdzić czy możemy zmniejszyć epsilon przy zachowaniu 3 różnych sąsiadów. W związku aby zrobić to optymalnie musimy zacząć od punktów A i L. Dla punktu A szacujemy odległość następująco: odległość(L,r) – odległość(F,r) musi być mniejsza od epsilona. 5,8-3,2=2,6. Jest to większe od 1,77 więc nie spełnia warunków. Ponadto z oszacowania wiemy że wszystkie punkty pod nim również w tym

przypadku nie będą dobre. W takim razie rozpatrując punkt L obliczamy oszacowanie: 3,2-1,8=1,4. Jest to mniejsze od 1,77. W związku z tym obliczamy rzeczywistą odległość równą 0,6. W związku z tym punkt C przestaje być sąsiadem punktu F i zamiast niego sąsiadem zostaje punkt L i nowa wartość epsilon to max(1,64, 1,25, 0,6) = 1,64. Więc musimy rozpatrywać następnie punkt K, dla którego oszacowanie to: 3,2 – 0,9 = 2,3 i jest to większe od epsilona więc nie rozpatrujemy już tego punku oraz jeśli byłyby jeszcze za nim kolejne punkty to również byśmy ich nie rozpatrywali. Więc znaleźliśmy 3^+-NN sąsiedztwo dla punktu F i są to punkty H,G,L.

Następnie musimy zliczyć dla każdego punktu ile razy występuje on w sąsiedztwie dla innych punktów dzięki czemu będziemy mieli wyznaczone odwrotne sąsiedztwo dla danego punktu. Teraz trzeba policzyć współczynnik gęstości NDF dla każdego punktu i jeżeli znajdziemy punkt dla którego NDF >= 1 wtedy taki punkt stanie się punktem rdzeniowym i wszyscy sąsiedzi dla niego są zapamiętywani i oznaczani jako ta dana grupa. Następnie dla każdego sąsiada z pamięci ponownie sprawdzamy NDF i jeżeli jest również rdzeniowy wtedy powtarzamy dla niego operacje. W przypadku gdy w pamięci nie ma już żadnych punktów to sprawdzamy czy jest jeszcze jakiś inny punkt który jest rdzeniowy a nie należy jeszcze do żadnej grupy. Jeżeli taki będzie to oznaczamy go nową wartością grupy i sprawdzamy sąsiedztwo tak jak w punkcie powyżej. W momencie gdy wszystkie punkty będą przyporządkowane do jakiejś grupy wtedy jest koniec działania algorytmu.

Opis implementacji

Przy implementacji algorytmu skorzystałem z języka C++, oraz biblioteki QT w celu lepszej interakcji użytkownika z programem. Cały zaimplementowany kod został zamieszczony w repozytorium na moim GitHubie [2].

W ramach projektu zaimplementowałem algorytm NBC w dwóch wariantach:

- Wykorzystujący nierówność trójkąta przy wyznaczaniu k^+-NN sąsiedztwa dla każdego punktu w formie odzwierciedlającej algorytm opisany w artykule.
- Nie wykorzystujący nierówności trójkąta naiwny algorytm wyznaczania wyznaczaniu k^+-NN sąsiedztwa dla każdego punktu o złożoności x^2 sprawdzający każdy punkt z każdym.

Ogólny schemat działania zaimplementowanego algorytmu można zauważyć w konstruktorze klasy NBC na rysunku 4.

```
openDataset(path, delimiter);
if (TIEnabled) {
    countDistance();
    sortPoints();
    setIndex();
    findNeighbors();
    countNdf();
    putLabelsOn();
}
else {
    setIndex();
    findNeighborsWithoutTI();
    countNdf();
    putLabelsOn();
}
```

Rysunek 4 Ogólny schemat działania algorytmu

Najpierw wczytujemy Dataset do pamięci w programu a następnie zapisuje wszystkie punkty do Vectora punktów gdzie będą przechowywane do dalszych obliczeń. W przypadku podstawowego algorytmu następnie wyznaczamy odległość za pomocą odległości Euklidesowej danego punktu do punktu referencyjnego, który został wyznaczony jako minimalne wartości przy wczytywaniu Datasetu. Następnie należy posortować Vector punktów za pomocą tej odległości. Następnie przydzielamy każdemu punktowi unikalny identyfikator w całym systemie.

Teraz przejdziemy do wyznaczenia sąsiadów każdego z punktów. W pierwszym kroku musimy wybrać k^{+}NN najbliższych sąsiadów aby mieć pewność że będziemy mieć przynajmniej k punktów w sąsiedztwie o promieniu epsilona. W tym celu korzystając z posortowanego Vectora porównujemy między sobą sąsiada z dołu z sąsiadem z góry i w zależności ile punktów potrzeba do sąsiedztwa to tyle razy wybieramy zawsze z tych punktów punkt o mniejszej odległości tak jak na rysunku 5.

```
void NBC::findNeighborsOfPoint(int point) {
    points[point].setMaxChecked(point);
    points[point].setMinChecked(point);

    for (int i = 0; i < k; i++) {
        double dist1 = getDistanceNextPoint(points[point].getMaxChecked() + 1);
        double dist2 = getDistancePrevPoint(points[point].getMinChecked() - 1);
        if (dist1 < dist2) {
            Neighbor n;
            n.index = points[point].getMaxChecked() + 1;
            n.realDistance = countRealDistanceToPoint(point, n.index);
            points[point].addNeighbor(n);
            points[point].incrementMaxChecked();
        }
        else {
            Neighbor n;
            n.index = points[point].getMinChecked() - 1;
            n.realDistance = countRealDistanceToPoint(point, n.index);
            points[point].addNeighbor(n);
            points[point].decrementMinChecked();
        }
    }
}
```

Rysunek 5 Wyznaczenie k najbliższych sąsiadów dla punktu

Za pomocą indeksów minChecked oraz maxChecked będzie wiadomo który z punktów zostały sprawdzone. Natomiast wybieramy zawsze ten punkt gdy różnica odległości do punktu referencyjnego jest najniższa. Wtedy odległość między tymi punktami potencjalnie może być najniższa, ale jest to tylko oszacowanie. W momencie wybrania punktu jako kandydata na sąsiada wtedy zapisujemy do Vectora sąsiadów dla danego punktu oraz odpowiednio aktualizujemy informacje o sprawdzonych już punktach. Więc teraz mamy gwarancję przynajmniej k punktów w sąsiedztwie dla każdego punktu.

Jednak nie mamy gwarancji że wybrani sąsiedzi są najbliższymi możliwymi sąsiadami dla danego punktu. W związku z tym w kolejnym kroku należy to sprawdzić. Na początku należy dla każdego punktu obliczyć Epsilon aby znać odległość najdalszego sąsiada do danego punktu. Następnie jak na rysunku 6 w pętli while sprawdzamy kolejne punkty mają w Vectorze punktów większą oszacowaną odległość. Jeżeli oszacowanie odległości zdefiniowane jako różnica odległości sprawdzanego punktu z odległością aktualnie sprawdzanego potencjalnego sąsiada. Jeżeli różnica jest większa od Epsilona wtedy możemy już zakończyć sprawdzanie sąsiadów w tym kierunku, ponieważ z oszacowania wiemy

że każdy kolejny punkt będzie coraz większy od Epsilona. Natomiast jeżeli obliczone oszacowanie jest mniejsze od Epsilona wtedy musimy obliczyć rzeczywistą odległość między punktami i jeżeli będzie ona mniejsza od Epsilona wtedy dany punkt staje się nowym sąsiadem a punkt o największej odległości spośród dotychczasowych sąsiadów przestaje być sąsiadem dla danego punktu. Jeżeli w tym przypadku rzeczywista odległość będzie większa od Epsilona to nie możemy zastąpić tu żadnego sąsiada jednak jeszcze nie mamy pewności czy kolejny punkt w Vectorze punktów nie będzie lepszy od jednego z już istniejących sąsiadów, więc nie przerywamy jeszcze pętli. W programie zdefiniowałem również drugą analogiczną funkcję sprawdzających punkty mające mniejszą obliczoną odległość od punktu referencyjnego, która wygląda bliźniaczo do zaprezentowanej tutaj funkcji na rysunku 6.

```
while (points[i].getMaxChecked() + 1 < points.size()) {
    int nextPoint = points[i].getMaxChecked() + 1;
    double estimatedDist = fabs(points[i].getDistance() - points[nextPoint].getDistance());
    if (estimatedDist < points[i].getEpsilon()) {
        double realDistance = countRealDistanceToPoint(i, nextPoint);
        if (realDistance < points[i].getEpsilon()) {
            Neighbor n;
            n.index = nextPoint;
            n.realDistance = realDistance;
            points[i].replaceNeighbor(n);
            points[i].incrementMaxChecked();
            points[i].countEpsilon();
        } else {
            points[i].incrementMaxChecked();
            continue;
        }
    }
    else {
        break;
    }
}
```

Rysunek 6 Znalezienie optymalnych odległościowo sąsiadów dla punktu

Następnie następuje obliczenie wartości NDF zgodnie ze wzorem: $= \frac{|Rk^{+}\text{-}NN(p)|}{k^{+}\text{-}NN(p)}$

Ostatnim krokiem jest przydział każdemu punktowi etykiety w zależności od tego do jakiej grupy został ten punkt zakwalifikowany. Szczegółowy algorytm przydziału etykiet został pokazany na rysunku 7.

```
groupIndex++;
seeds.push_back(i);
while (!seeds.empty()) {
    points[seeds[0]].setLabel(groupIndex);
    for (int j = 0; j < points[seeds[0]].getNeighborsSize(); j++) {
        int neighborIdx = points[seeds[0]].getNeighbor(j).index;
        if (points[neighborIdx].getLabel() <= 0) {
            points[neighborIdx].setLabel(groupIndex);
            if (points[neighborIdx].getNdf() >= NBCthreshold) {
                seeds.push_back(neighborIdx);
            }
        }
    }
    seeds.pop_front();
}
```

Rysunek 7 Przydział punktów do poszczególnych grup

Na początku pierwszy punkt dodajemy do Vectora seeds. Następnie dopóki w seeds znajduje się jakiś element to pierwszemu punktowi przydzielamy aktualny numer, następnie rozpatrujemy jego sąsiadów i jeżeli dotychczas dany sąsiad nie miał żadnego oznaczenia lub był szumem staje się częścią bieżącej grupy i dodajemy go też do Vectora seeds. W przypadku gdy w Vectorze seeds skończą się elementy wtedy inkrementujemy numer grupy i szukamy kolejnego elementu który nie został jeszcze przydzielony do żadnej grupy i wtedy powtarzamy dla niego powyższe czynności. W przypadku gdy jest uruchamiany algorytm NBC bez wykorzystania nierówności trójkąta wtedy omijamy kroki obliczania dystansu do punktu referencyjnego oraz sortowania elementów w pamięci. Natomiast zamiast nich znajdujemy sąsiadów według algorytmu przedstawionego na rysunku 8.

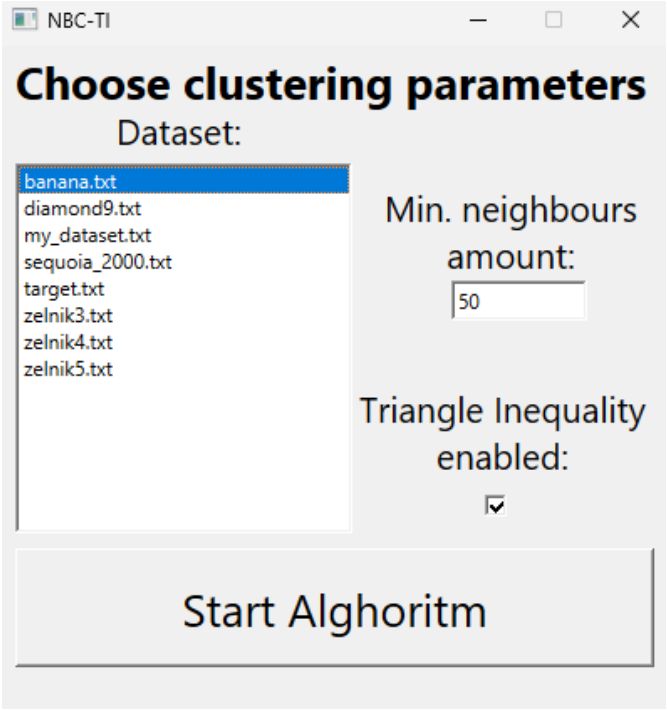
```
void NBC::findNeighborsWithoutTI() {
    for (int point = 0; point < points.size(); point++) {
        for (int j = 0; j < k; j++) {
            Neighbor n;
            n.index = -100;
            n.realDistance = std::numeric_limits<double>::max();
            points[point].addNeighbor(n);
        }
        points[point].countEpsilon();
        for (int j = 0; j < points.size(); j++) {
            if (point == j) continue;
            double realDistance = countRealDistanceToPoint(point, j);
            if (realDistance < points[point].getEpsilon()) {
                Neighbor n;
                n.index = j;
                n.realDistance = realDistance;
                points[point].replaceNeighbor(n);
                points[point].countEpsilon();
            }
        }
    }
}
```

Rysunek 8 Wyszukiwanie sąsiadów bez wykorzystania nierówności trójkąta

Również wykonujemy to w dwóch krokach. Najpierw wybieramy k pierwszych punktów aby mieć na pewno k punktów w sąsiedztwie Epsilonowym, a następnie algorytmem naiwnym sprawdzamy odległości każdego punktu z każdym aby mieć pewność że w sąsiedztwie znajdują się najbliższe możliwe punkty. Więc algorytm ten ma złożoność n^2, co w przypadku bardzo dużych zbiorów danych może powodować problemy wydajnościowe.

Instrukcja użytkownika

W momencie uruchomienia programu zostanie wyświetlone okno widoczne na rysunku 9 w którym należy zdefiniować parametry algorytmu grupowania.

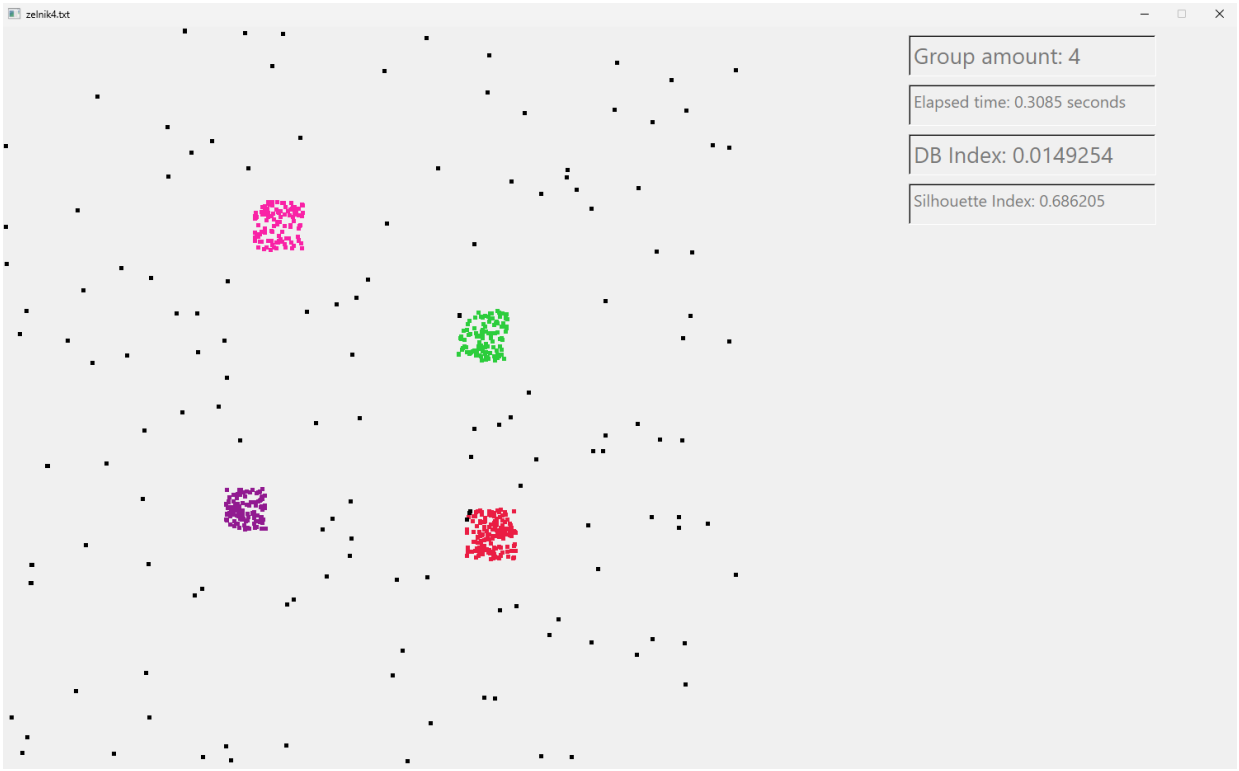


Rysunek 9 Okno startowe programu

Na oknie użytkownik może wybrać jeden ze zdefiniowanych wcześniej datasetów albo do folderu data może umieścić własny dataset który po uruchomieniu programu będzie widoczny na liście. Dataset powinien mieć następujący format: „X,Y,” czyli dwie współrzędne rozdzielone przecinkami. Liczba wierszy może być dowolna jednak nie mniejsza od minimalnej liczby sąsiadów.

Oprócz tego należy zdefiniować właśnie minimalną liczbę sąsiadów którzy będą szukani dla każdego punktu. A także można zdecydować czy algorytm ma wykorzystywać nierówność trójkąta czy nie.

Po wybraniu datasetu „zelnik4.txt”, 50 sąsiadów i zaznaczonego checkboxa TI widać następujący rezultat na rysunku 10.



Rysunek 10 Wygląd okna z rezultatem grupowania

Zgodnie z rysunkiem 10 widać na różne kolory zaznaczone grupy. Kolorem czarnym został oznaczony szum. Oprócz tego po prawej stronie widać po każdym uruchomieniu algorytmu jaka jest łączna liczba grup, ile czasu zajęło grupowanie danych oraz wartości dwóch wskaźników: Davies-Bouldin oraz Silhouette.

Zbiory danych

Do testowania napisanego programu wykorzystałem na początku prosty Dataset z przykładu zaprezentowanego w artykule, a następnie zastosowałem algorytm grupowania do stworzonych zbiorów zamieszczonych na GitHubie [3]. Spośród nich wybrałem następujące zbiory:

- Banana
- Diamond9
- Target
- Zelnik3
- Zelnik4
- Zelnik5

Szczegółowy wygląd powyższych zbiorów zostanie zaprezentowany w rozdziale poniżej podczas eksperymentów.

Wyniki eksperymentów pokazujących właściwości proponowanego rozwiązania

Podczas ewaluacji grupowania dla każdego zbioru danych będzie brane pod uwagę na początku ocena przyporządkowania do grup na podstawie obrazu wyjściowego. Następnie będzie brany pod uwagę łączny czas potrzebny na przydział poszczególnych punktów do grup. Oprócz tego grupowanie będzie

oceniane za pomocą dwóch niezależnych od siebie wewnętrznych miar ewaluacji na rysunkach 11 i 12.

Wewnętrzna miara ewaluacji grupowania
Davies-Bouldin

Miara ewaluacji grupowania Davies-Bouldin:

$$Davies\text{–}Bouldin = \frac{1}{n}\sum_{i=1}^{n} \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)} \right), \text{ gdzie}$$

- n – liczba odkrytych grup,
- c_k – centroid k–tej grupy,
- σ_k – średnia odległość elementów k-tej grupy do jej centroidu c_k,
- $d(c_i, c_j)$ – odległość pomiędzy centroidami c_i, c_j.

Rysunek 11 Wzór obliczający miarę Daviesa-Bouldina - źródło wykład

Wewnętrzna miara ewaluacji grupowania
Silhouette – wskaźnik sylwetkowy

- Ewaluacja grupowania dla punktu i:

$$S(i) = \frac{b(i) - a(i)}{\max\{b(i), a(i)\}}, \text{ gdzie}$$

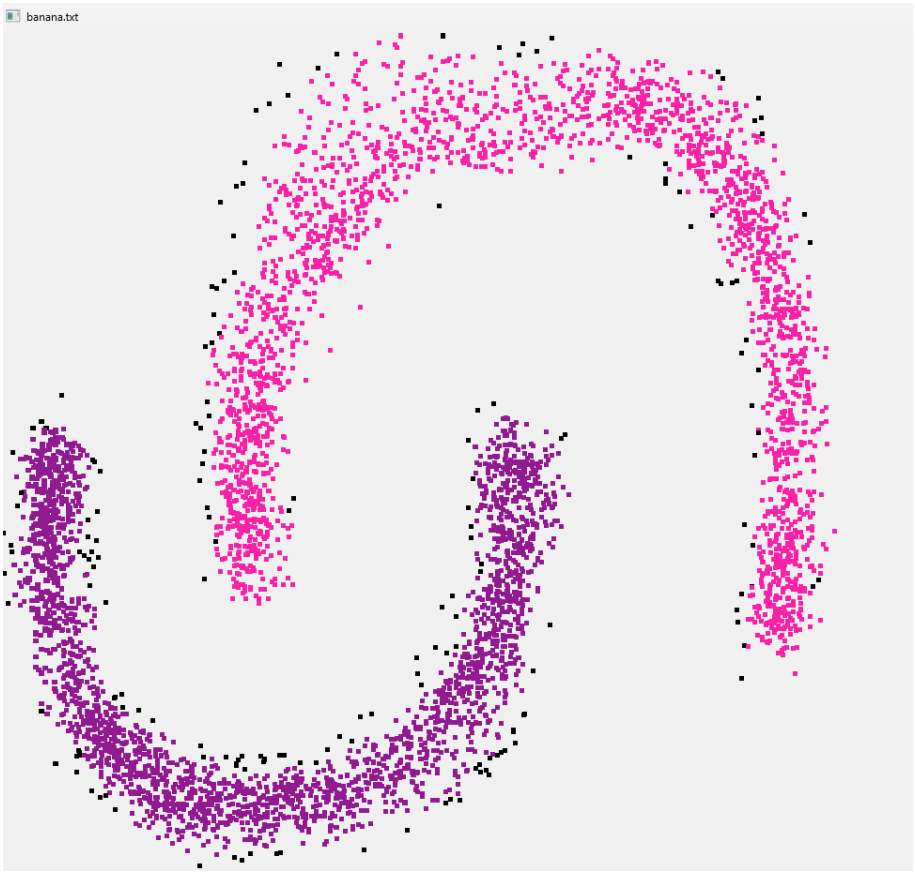
  - $a(i)$ – średnia odległość od punktu i do innych punktów w grupie punktu i,
  - $b(i)$ – najmniejsza średnia odległość od punktu i do wszystkich punktów grupy, która nie zawiera punktu i.

- Ewaluacja dla grupy – średnia wartość $S(i)$ dla punktów tej grupy.
- Ewaluacja grupowania całego zbioru danych – średnia wartość $S(i)$ dla wszystkich punktów zbioru danych.

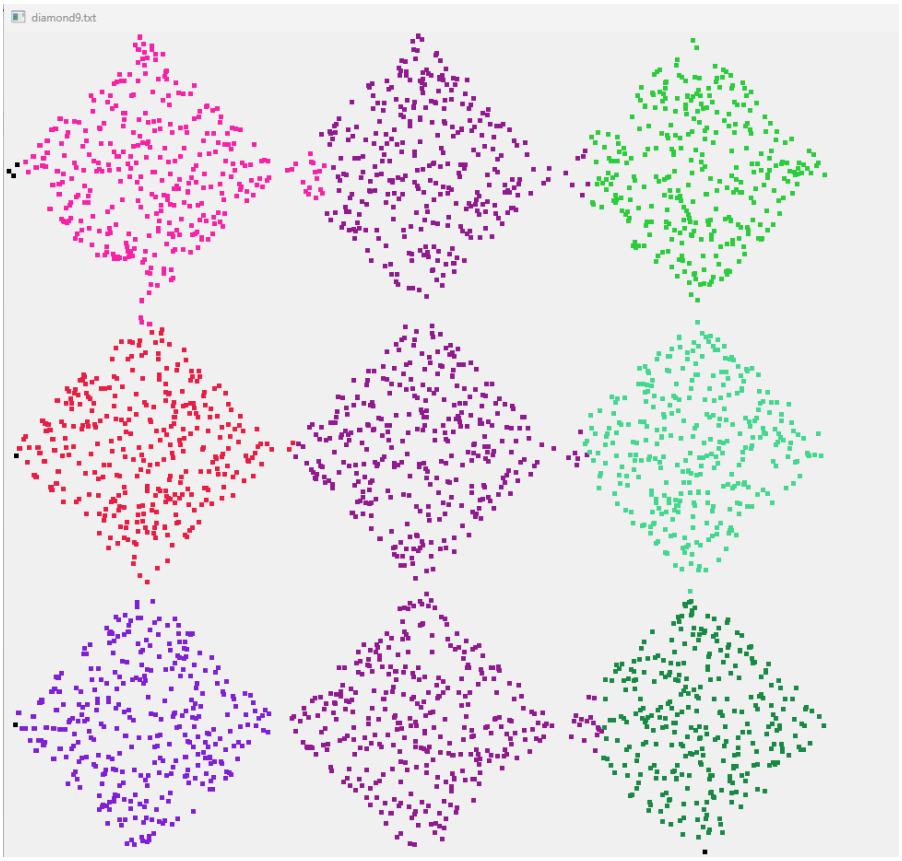
Rysunek 12 Wzór obliczający miarę Silhouette - źródło wykład

Warto zaznaczyć że wartość wskaźnika Daviesa-Bouldina przyjmuje wartości <0, ∞> gdzie im lepsze grupowanie tym wartość jest bliższa 0. Natomiast wskaźnik Silhouette przyjmuje wartości <-1, 1> gdzie najlepsze grupowanie jest dla wartości 1. Oprócz tego każdy z testów zostanie wykonany ponownie dla algorytmu niekorzystającego z nierówności trójkąta przy znajdywaniu sąsiadów.

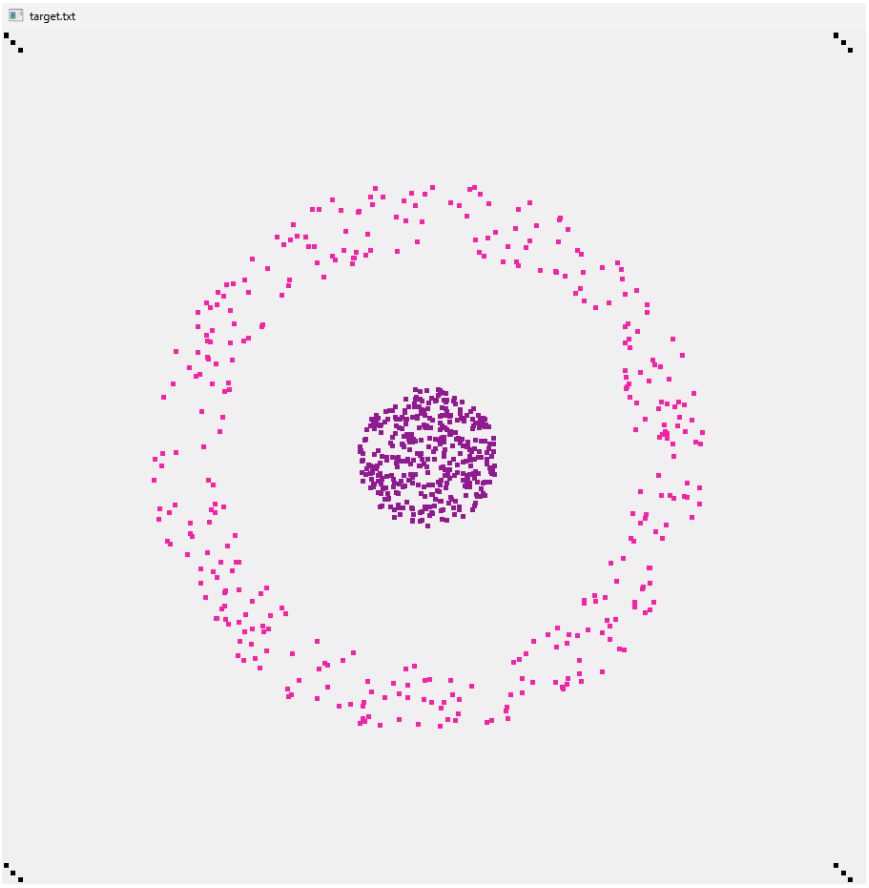
Poniżej znajdują się wyniki ewaluacji:



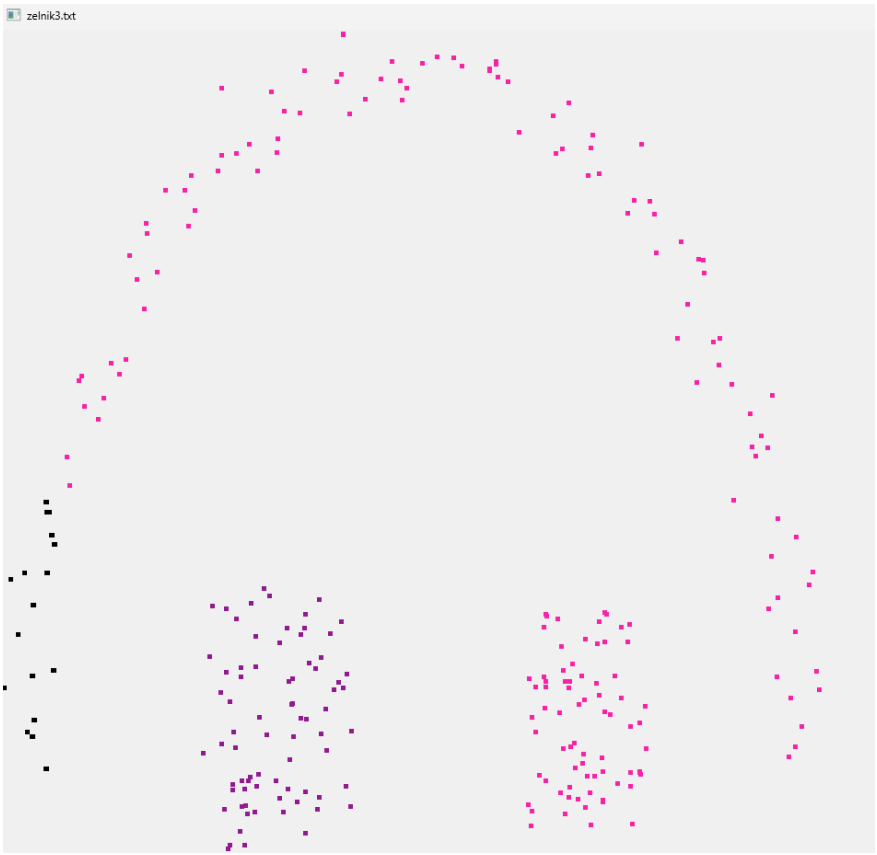
Rysunek 13 Zbiór Banana



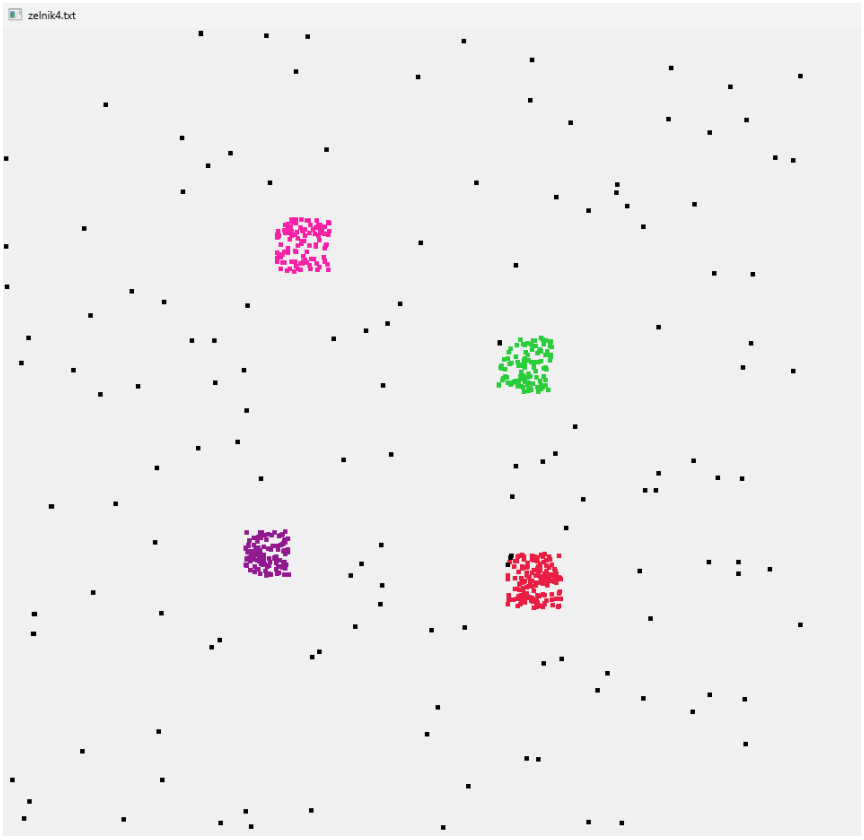
Rysunek 14 Zbiór Diamond9



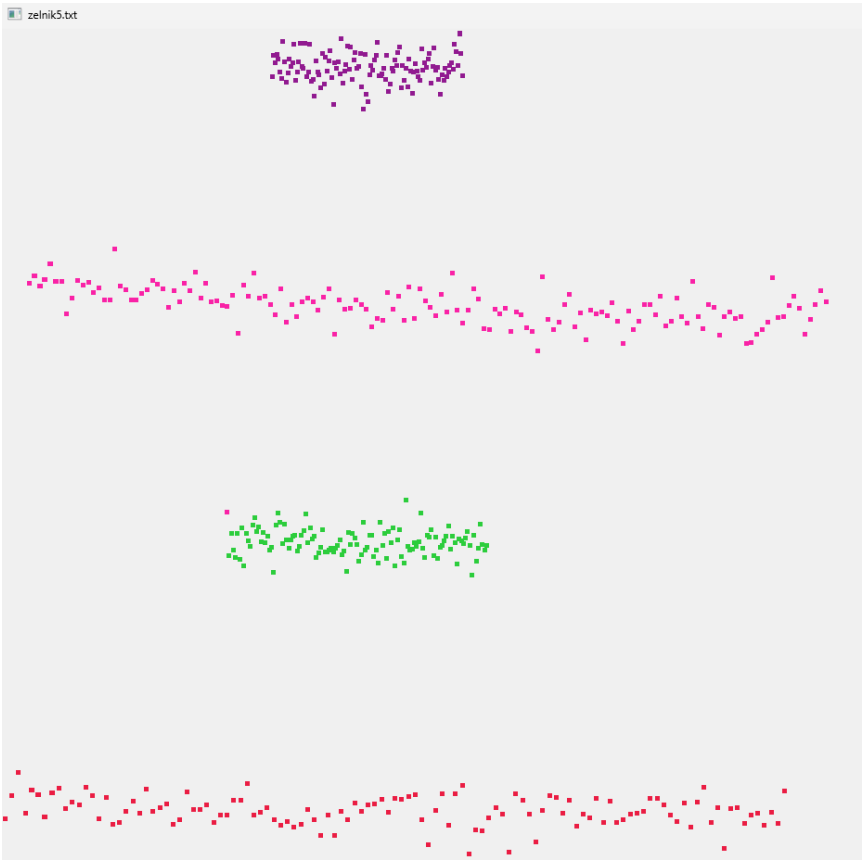
Rysunek 15 Zbiór Target



Rysunek 16 Zbiór Zelnik3



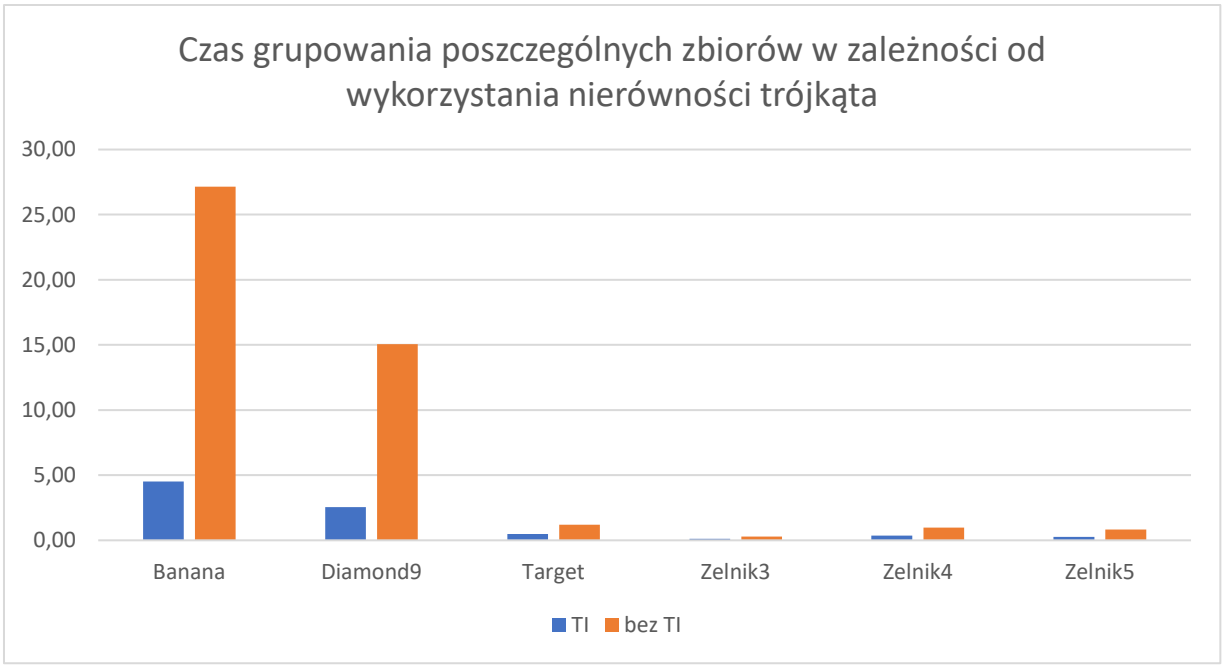
Rysunek 17 Zbiór Zelnik4

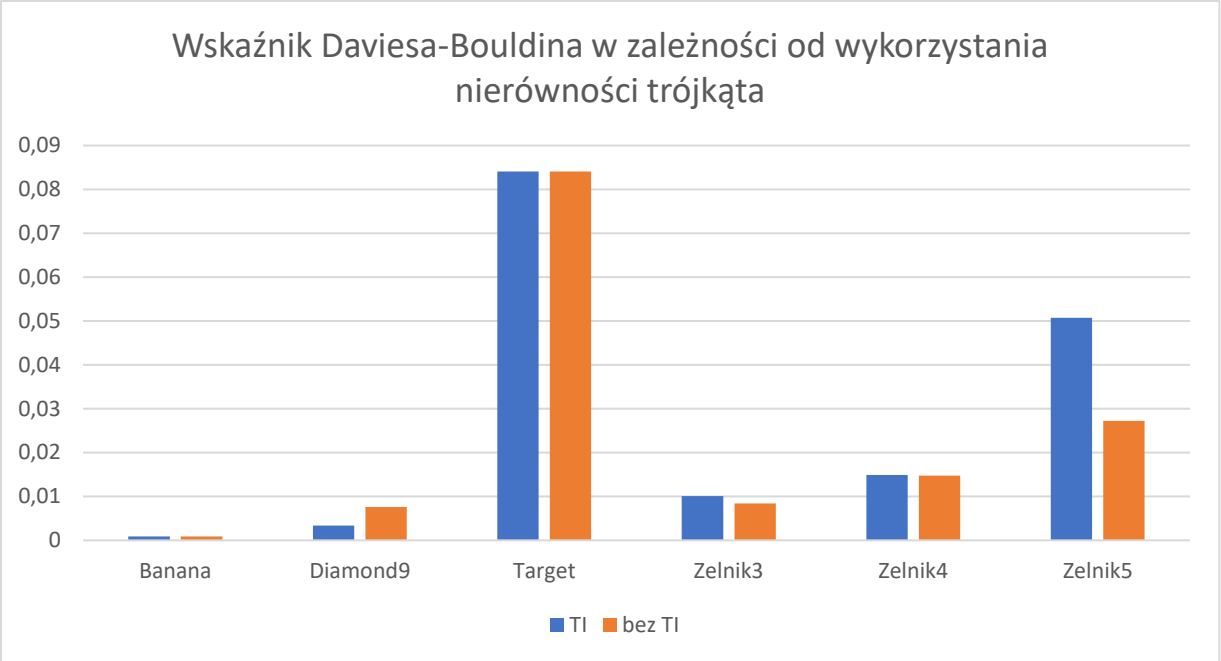


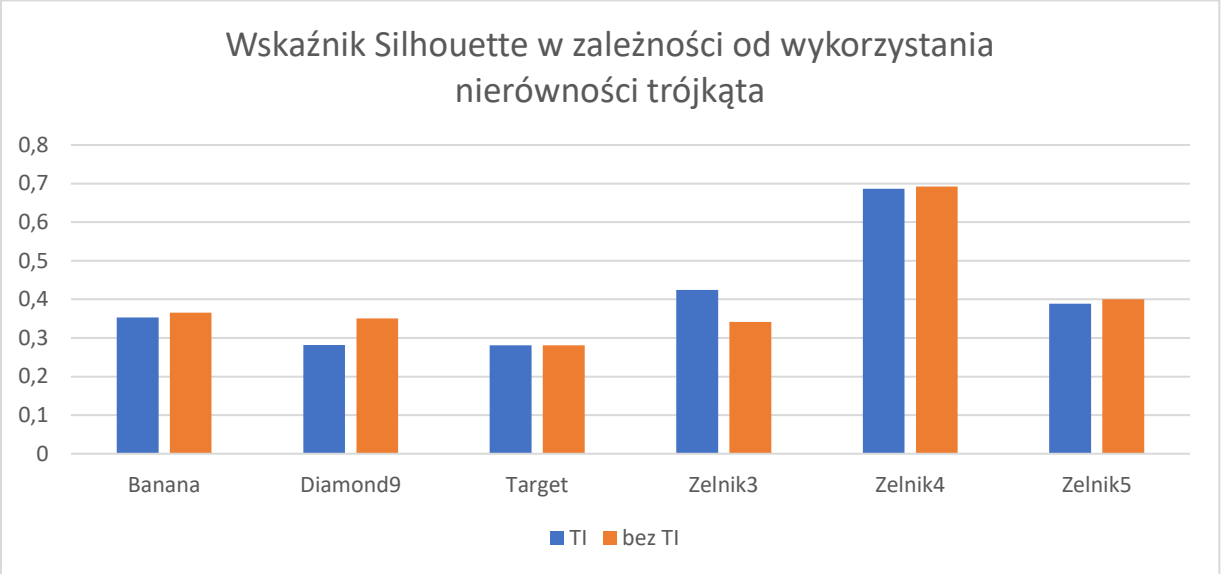
Rysunek 18 Zbiór Zelnik5

Oprócz tego szczegółowe wyniki testów zebrałem do tabeli:

Numer testu	Nazwa Zbioru	Wykorzystanie nierówności trójkąta	Liczba grup	Czas grupowania [s]	Wskaźnik DB	Wskaźnik Silhouette
1	Banana	Tak	2	4.5059	0.00091	0.35322
2		Nie	2	27.1471	0.00087	0.36570
3	Diamond9	Tak	5	2.5442	0.00334	0.28195
4		Nie	7	15.0508	0.00764	0.35027
5	Target	Tak	2	0.4881	0.08411	0.28132
6		Nie	2	1.1973	0.08411	0.28132
7	Zelnik3	Tak	2	0.1169	0.01009	0.42476
8		Nie	2	0.2933	0.00843	0.34191
9	Zelnik4	Tak	4	0.3534	0.01492	0.68620
10		Nie	4	0.9727	0.01477	0.69223
11	Zelnik5	Tak	4	0.2756	0.05074	0.38857
12		Nie	4	0.8367	0.02722	0.40061







Wnioski

Po analizie wyników można zauważyć że dla każdego zbioru wykorzystanie nierówności trójkąta daje 3-5 razy lepsze czasy wykonania algorytmu w przeciwieństwie do naiwnego wyszukiwania sąsiadów. Nie ma w tym nic dziwnego ponieważ przy zastosowaniu nierówności trójkąta możemy z góry pesymistycznie oszacować że wiele punktów nie może być sąsiadami danego punktu, podczas gdy w drugim przypadku mamy złożoność n^2, ponieważ dla każdego punktu wyszukujemy najlepszych spośród wszystkich dostępnych punktów w zbiorze danych. Ponadto efekty grupowania patrząc na obrazy wynikowe wyglądają bardzo dobrze, co potwierdzają wskaźniki jakości grupowania Daviesa-Bouldina oraz Silhouette. Zdecydowanie algorytm z artykułu lepiej sobie radzi w przypadku gdy poszczególne grupy są od siebie w jakiś sposób oddzielone, ponieważ w przypadku zbioru Diamond9 gdy poszczególne grupy są w bardzo bliskiej odległości to dla ustalonej przeze mnie początkowo wartości ilości minimalnej liczby sąsiadów na 50 algorytm wykrywa 7 różnych grup. Natomiast te 9 grup który wydawałoby się że są poprawną ilością można uzyskać dostrajając ręcznie tę wartość ilości sąsiadów na 15.

Bibliografia

[1] M. K. a. P. Lasek, „A Neighborhood-Based Clustering by Means of,” WarsawSpringer, 2010.

[2] https://github.com/wshknmt/NBC-TI

[3] https://github.com/deric/clustering-benchmark

Spis treści

Spis rysunków