Miniprojekt IV

Metody probabilistyczne w uczeniu maszynowym

Łukasz Trzos

0 Temat projektu

Udostępnionych jest 10 plików z danymi.

1. Plik **rp.data** to plik znany już z drugiego miniprojektu. Zawiera on dane o 9 cechach oraz przypisaną im klasę (2 lub 4).
2. Plik **dane_18D.txt** zawiera dane o 18 cechach, jednak bez przypisanych klas.
3. Pliki **dane_2D_n.txt** ($n = 1, ..., 8$) zawierają współrzędne punktów na płaszczyźnie oraz przypisaną im klasę. Liczba klas jest różna w różnych plikach.

Dla każdego z powyższych zbiorów danych porównaj działanie następujących trzech algorytmów klasteryzacji:

- algorytm k-średnich (*k-means*) lub k-średnich++ (*k-means++*),
- grupowanie hierarchiczne (*hierarchical clustering*),
- klasteryzacja spektralna (*spectral clustering*).

Jeśli uznasz to za pomocne, zawsze możesz korzystać z funkcji jądrowych.

W każdym przypadku poza danymi z pliku **rp.data** uzasadnij decyzję o wybranej liczbie klastrów. W przypadku danych z plików **dane_2D_n.txt** sprawdź, czy zwracasz liczbę klastrów równą liczbie zadanych klas. Znając liczbę klastrów, sprawdź jakość każdego algorytmu klasteryzacji obliczając błąd klasyfikacji.

Zwizualizuj (np. używając różnych kolorów) klastry po zastosowaniu algorytmów dla danych z plików **dane_2D_n.txt**.

Na podstawie uzyskanych wyników wyciągnij wnioski: który algorytm Twoim zdaniem sprawdził się najlepiej w każdym z przypadków?

Do miniprojektu dołączone są dwa artykuły dotyczące klasteryzacji spektralnej oraz algorytmu *k-means++*:

- David Arthur, Sergei Vassilvitskii
  ***k-means++**: The Advantages of Careful Seeding,*
- Andrew Ng, Michael Jordan, Yair Weiss
  On Spectral Clustering: Analysis and an algorithm.

1 Opisy teoretyczne algorytmów klasteryzacji

1.1 K-means

Algorytm na wejściu otrzymuje liczbę klastrów, na jaką ma podzielić zbiór danych. Będziemy szukać takiego podziału, żeby dla każdego punktu centroid klastra, do którego należy był mu najbliższy spośród centroidów wszystkich klastrów. Centroid danego zbioru punktów to punkt o współrzędnych będących średnią arytmetyczną współrzędnych wszystkich punktów w zbiorze.

Algorytm jest iteracyjny, dla danego wyboru centroidów będziemy starać się go poprawić aż uzyskamy rozwiązanie. Strategie inicjalizacyjne różnią się w zależności od wersji algorytmu:

- W standardowym algorytmie **k-means** wybieramy po prostu losowe k punktów z jednostajnym prawdopodobieństwem ze zbioru danych
- W rozszerzonej wersji, **k-means++**, faworyzujemy wybory takich punktów, które leżą wystarczająco daleko od siebie. Na początku wybieramy pierwszy punkt losowo. Następnie w każdej z $k-1$ iteracji dla każdego pozostałego punktu x liczymy jego odległość od najbliższego mu centroidu $D(x)$, a następnie losujemy punkt z rozkładem ważonym, w którym prawdopodobieństwo wybrania punktu x jest proporcjonalne do $D(x^2)$.

Poniżej przedstawiam obydwie te strategie zaimplementowane w Pythonie:

```python
centroids = []

if not isPlusPlus:
    #k-means
    centroids = random.sample(points, k)
else:
    #k-means++
    centroids.append(random.choice(points))

    for _ in range(1, k):
        distances = []

        for p in points:
            min_dist_sq = min(utils.EuclideanDistSquared(p,c) for c in centroids)
            distances.append(min_dist_sq)

        total = sum(distances)
        probs = [d / total for d in distances]
        cumulative_probs = np.cumsum(probs)

        r = random.random()
        for i, cp in enumerate(cumulative_probs):
            if r < cp:
                centroids.append(points[i])
                break
```

Rysunek 1: Strategie inicjalizacji centroidów w algorytmach k-means

Każdy wybrany w taki sposób centroid dostaje przypisane ID klastru. Następnie wykonujemy następujący algorytm:

- Dla każdego punktu ze zbioru przypisujemy mu klaster najbliżego mu centroidu
- Obliczamy nowe centroidy dla każdego z klastrów będące średnią po współrzędnych wszystkich punktów w danym klastrze

Powtarzamy powyższe kroki aż uzyskamy sytuację, w której podczas iteracji żaden punkt nie zmieni swojego klastru. Algorytm zawsze będzie zbiegał do takiej sytuacji, ponieważ w każdym kroku ściśle zmniejszamy sumę odległości punktów od przypisanych im centroidów (**WCSS**), a ilość możliwych podziałów punktów jest skończona. Nie zawsze uda nam się jednak uzyskać najlepsze rozwiązanie, wynika to wprost z faktu, że początkowe centroidy są losowane.

```python
pointsClusters = [0] * len(points)

    pointChangedClasses = True

    pointKeyFrames = []
    centroidKeyFrames = []

    while(pointChangedClasses):
        pointChangedClasses = False

        #Classify points
        #If anything changed in their classification, the loop should continue
        for i, point in enumerate(points):
            currentMinDist = math.inf
            currentClosestCentroid = -1

            for j, centroid in enumerate(centroids):
                dist = math.sqrt(utils.EuclideanDistSquared(point, centroid))

                if(dist < currentMinDist):
                    currentMinDist = dist
                    currentClosestCentroid = j

            if(pointsClusters[i] != currentClosestCentroid):
                pointChangedClasses = True
            pointsClusters[i] = currentClosestCentroid

        if isVisualized:
            pointKeyFrames.append(list(pointsClusters))
            centroidKeyFrames.append(list(centroids))

        #Change centroids
        for cluster in range(k):
            pointsInOurCluster = [p for i, p in enumerate(points) if pointsClusters[i]==cluster]

            if pointsInOurCluster:
                newCentroid = np.mean(pointsInOurCluster, axis=0)
            else:
                newCentroid = random.choice(points) #Random reinicialization

            centroids[cluster] = newCentroid
```

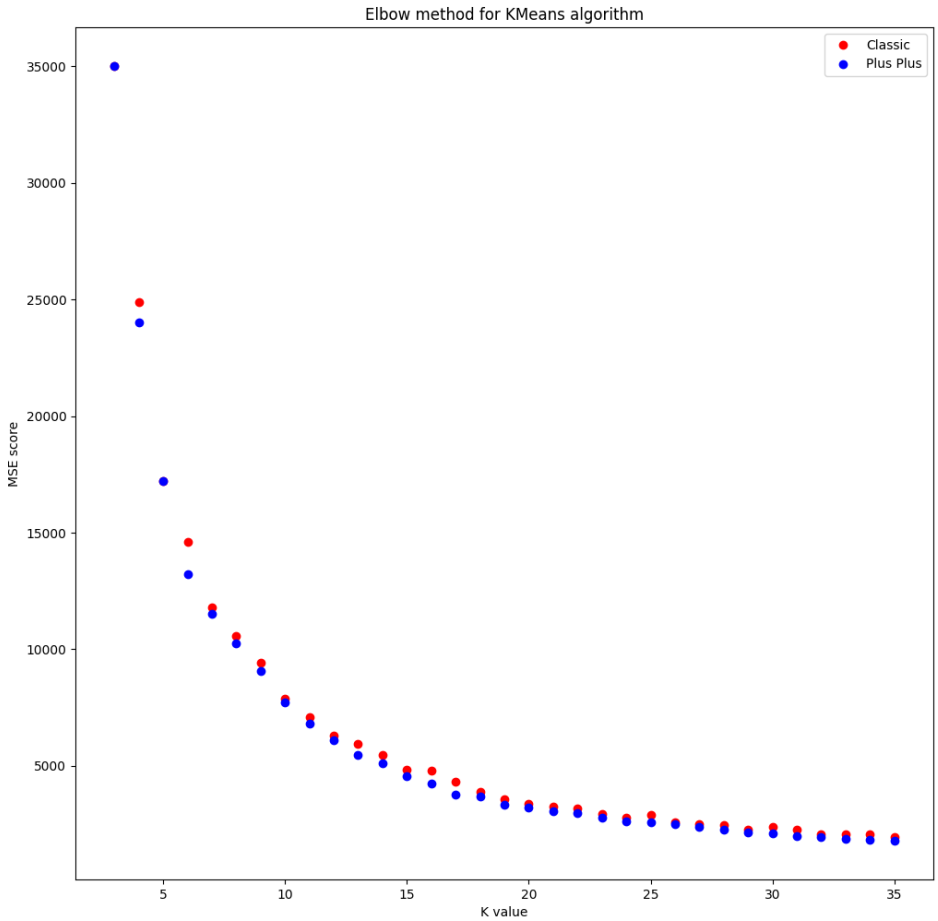
Rysunek 2: Implementacja pętli algorytmu k-means

Warto dodać, że w niektórych przebiegach, zwykle dla większych wartości parametru k, może wystąpić sytuacja, że centroid nie otrzyma żadnego punktu i klaster "zniknie". W takiej sytuacji reinicjalizuję go do losowo wybranego punktu ze zbioru. Trochę czasu zajęło mi zauważenie, dlaczego kod nie zawsze zwraca taką liczbę klastrów o jaką został poproszony.

Ważną cechą algorytmu jest to, że musimy od góry podać docelową liczbę klastrów. Często jednak otrzymujemy dane, dla których ani ona, ani nawet jej przybliżenie, nie jest znana.

Z pomocą przychodzi metoda łokcia. Polega ona na wywołaniu algorytmu na różnych wartościach parametru k, dla każdej otrzymanej klasteryzacji obliczamy wspomniany wcześniej wskaźnik **WCSS**, a następnie przedstawiamy te dane na wykresie. Szukamy na wykresie punktu, w którym spadek wskaźnika staje się minimalny, czyli następuje wypłaszczenie krzywej. Zazwyczaj przed tym obserwujemy znacznie ostrzejszy spadek. Taki punkt nazywamy "punktem łokcia". Wybieramy odpowiadającą mu liczbę klas ponieważ, jak zaobserwowaliśmy, dodawanie dodatkowych klas nie przynosi już znacznej poprawy.

Na kolejnej stronie widoczna jest wspomniana zależność dla przykładowego zbioru danych dla $k \in [3, 35]$. Na jej podstawie możnaby szacować, że prawdopodobnie oryginalny zbiór jest podzielony na 6 lub 7 klastrów, po tym spadek staje się wyraźnie łagodniejszy.



Rysunek 3: Przykładowy wykres powstały przy pomocy metody łokcia

Warto zaznaczyć, że celem metody nie jest minimalizacja wartości wskaźnika **WCSS**. W szczególności, wskaźnik ten wynosi 0 jeśli przyjmiemy, że klastrów jest tyle ile punktów, co oczywiście nie jest najlepszym wyborem.

Metoda ta nie zawsze daje jednak jasne wyniki, często charakterystyczny punkt na wykresie nie jest aż tak widoczny, przez co różne osoby mogłyby wskazać różne optymalne ilości klastrów. Dla danego podziału na klastry możemy policzyć również współczynnik sylwetki: Dla danego punktu x_i współczynnik wyraża się wzorem:

$$s(i) = \frac{b(i) - a(i)}{max\{b(i), a(i)\}}$$

gdzie:

- $a(i)$ oznacza średnią odległość punktu x_i od pozostałych punktów w jego klastrze
- $b(i)$ oznacza minimum po wszystkich pozostałych klastrach ze średniej odległości x_i od punktów w danym klastrze

Z definicji widać, że dla dowolnego punktu zachodzi $-1 \leqslant s(i) \leqslant 1$. Wartości bliskie 1 oznaczają, że dany punkt jest prawdopodobnie umiejscowiony w dobrym klastrze, natomiast wartości bliskie -1 świadczą o jego nieprawidłowej klasyfikacji.

Obliczenie tego wyniku zajmuje jednak $\mathcal{O}(n^2)$ operacji, co jest o wiele bardziej kosztowne niż sam algorytm. Możemy skorzystać z tego, że nasza zastosowana implementacja zwraca również pozycje centroidów i obliczyć lekko zmodyfikowany współczynnik $s'(i) = \frac{b'(i)-a'(i)}{max\{b'(i),a'(i)\}}$, gdzie

- $a'(i)$ to odległość punktu x_i od centroidu jego klastra
- $b'(i)$ to minimum po odległościach punktu x_i od centroidów pozostałych klastrów

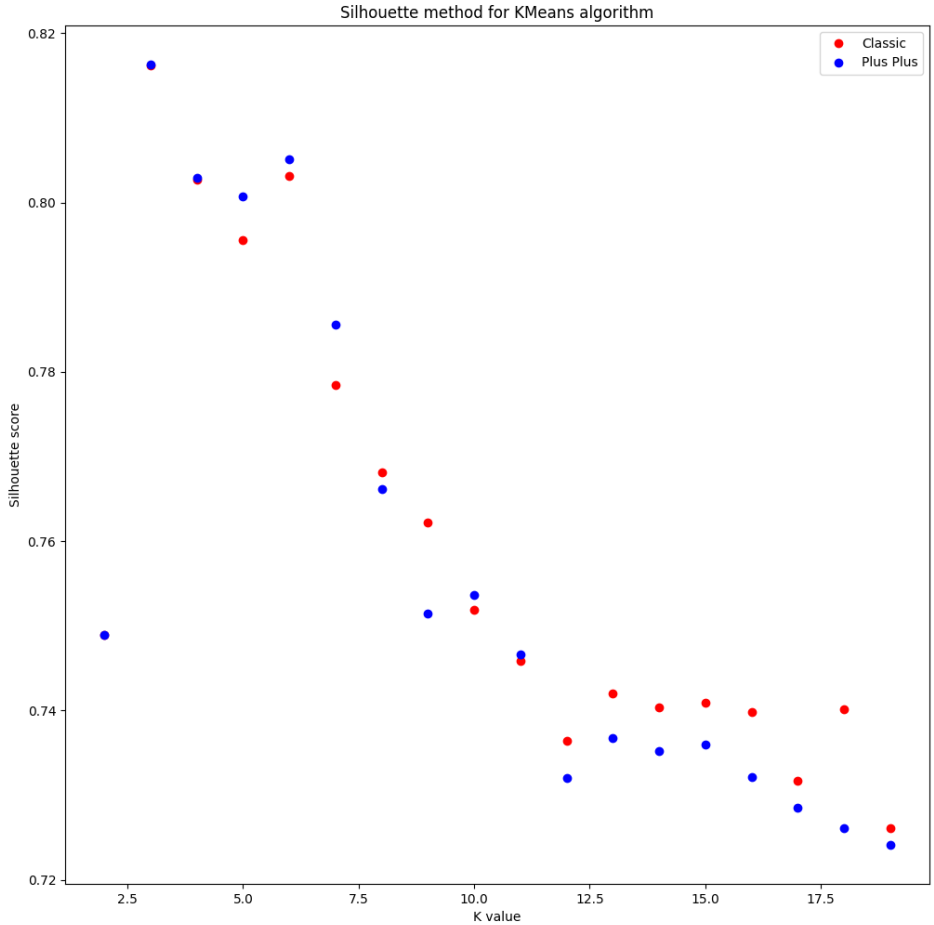
Zmodyfikowany współczynnik możemy policzyć w czasie $\mathcal{O}(kn)$.

```
pointsClusters, centroids = KmeansClassifyPoints(points, points, k, isVisualized=False, isPlusPlus=True)

a = [utils.EuclideanDistSquared(points[i], centroids[pointsClusters[i]]) for i in range(len(points))]
b = [min(utils.EuclideanDistSquared(points[i], centroids[j]) for j in range(len(centroids)) if j!=pointsClusters[i]) for i in range(len(points))]

SWithPlus += (1/len(points))*sum((b[i]-a[i])/(max(b[i], a[i])) for i in range(len(points)))
```

Rysunek 4: Implementacja uproszczonej metody sylwetkowej

Zaletą współczynnika sylwetki w stosunku do WCSS jest to, że przy wyborze liczby klastrów nie kierujemy się już intuicją związaną z analizą wykresu. Chcemy zmaksymalizować średnią wartość $s(i)$ na całym zbiorze punktów, zatem wybierzemy po prostu k, dla którego ta wartość jest największa.

Dla przykładu przedstawiam wykres zależności wartości współczynnika od liczby docelowych klastrów obliczony na tym samym zbiorze danych, co poprzedni:



Rysunek 5: Przykładowy wykres powstały przy pomocy metody sylwetkowej

Jak widać wyniki są nieco bardziej poszarpane. Zadziwiająco, najlepszy wynik uzyskujemy wybierając 3 klastry, ale drugie ekstremum znajduje się już w punkcie $k = 6$, czyli utwierdza nas to w przekonaniu, że powinniśmy wybrać ten wskaźnik. Prawdopodobnie wynika to z faktu, że algorytm k-means najlepiej radzi sobie dla sferycznych klas, a dane, na których prowadziliśmy testy, zawierały ich inne kształty, np. bardziej podłużne klastry.

Rozsądnym wyborem może być korzystanie z własności obydwu współczynników, zwłaszcza, że ich obliczenie nie jest szczególnie bardziej kosztowne niż sam algorytm, i wyznaczenie optymalnego k na podstawie ich analizy. W naszym przykładzie, zdecydowałbym się na podzielenie zbioru na 6 klastrów.

1.2 Grupowanie hierarchiczne

W przeciwieństwie do poprzedniego algorytmu, nie będziemy tym razem z góry przewidywać liczby rozłącznych klastrów w zbiorze danych. Stworzymy dendrogram ilustrujący proces stopniowego łączenia punktów w coraz większe grupy, który ostatecznie zakończymy, gdy pozostanie jeden klaster. Wtedy będziemy mogli co "uciąć" w dowolnym miejscu otrzymując tym samym podział na dowolną liczbę klastrów.

Algorytm w wersji aglomeracyjnej wygląda następująco: Na początku, tworzymy dla każdego punktu osobny klaster. W każdej iteracji pętli wybieramy 2 klastry, które są najbliżej siebie, a następnie łączymy w jeden. Zapisujemy do historii połączeń w celu późniejszego ucinania. Powtarzamy ten krok, aż sklasyfikujemy wszystkie punkty do jednego zbioru. Oczywiście, musimy ustalić, co oznacza termin "najbliższe klastry". Tutaj możemy zdefiniować wiele metryk. Stosuje się metryki oparte o 2 najbliższe lub najdalsze punkty, odległości centroidów, średnie odległości. W naszym projekcie skorzystam z szeroko stosowanej metody Warda:

Dla dwóch klastrów A oraz B odległość obliczana za pomocą metody Warda wynosi:

$$\text{WardDistance}(A, B) = \frac{|A||B|}{|A| + |B|}||\mu_A - \mu_B||_2$$

gdzie μ_A oraz μ_B to współrzędne centroidów zadanych klastrów.

Istnieje również deglomeracyjna wersja algorytmu, w której zaczynamy z jednym klastrem i dzielimy punkty do momentu, aż każdy będzie w osobnym klastrze. Wyznaczanie miejsca podziału klastrów jest jednak znacznie trudniejsze obliczeniowo niż iteracyjne łączenie dwóch najbliższych. Złożoność obliczeniowa jest również większa, dlatego w praktyce niemal zawsze korzysta się z metody aglomeracyjnej.

Omówimy krótko strategię implementacji. Możemy używać jednego kopca w celu dostępu do obecnych najkrótszych dystansów. Gdy łączymy dwa klastry, po prostu przypisujemy ich połączeniu zupełnie nowe ID, dzięki czemu nie musimy wyciągać nic z kopca. Musimy tylko pamiętać, które numery klastrów były już łączone z czymkolwiek. W ten sposób możemy uzyskać złożoność algorytmu $\mathcal{O}(n^2 log(n))$, gdzie n to liczba punktów w zbiorze danych.

```python
def HierarchicalClassifyPoints(originalPoints, points, isVisualized, dataID=0):

    isAlreadyConnected = [False for _ in range(len(points))]

    clusterCenters = [point for point in points]
    clusterSizes = [1 for _ in range(len(points))]

    connectionHistory = []

    H = []

    for i in range(len(points)):
        for j in range(i+1, len(points)):
            heapq.heappush(H, (WardDistance(clusterCenters[i], clusterCenters[j], 1, 1), (i,j)))

    currentNewClusterID = len(points)
    currentIteration = 0

    while H:
        (D, (i, j)) = heapq.heappop(H)

        if not isAlreadyConnected[i] and not isAlreadyConnected[j]:
            #Connect 2 clusters together
            isAlreadyConnected[i] = True
            isAlreadyConnected[j] = True

            isAlreadyConnected.append(False)
            clusterCenters.append(list((np.array(clusterCenters[i]) * clusterSizes[i] + np.array(clusterCenters[j]) * clusterSizes[j])/(clusterSizes[i] + clusterSizes[j])))
            clusterSizes.append(clusterSizes[i] + clusterSizes[j])

            connectionHistory.append(((i,j), currentIteration, D))

            for k in range(currentNewClusterID):
                if not isAlreadyConnected[k]:
                    heapq.heappush(H, (WardDistance(clusterCenters[k], clusterCenters[currentNewClusterID], clusterSizes[k], clusterSizes[currentNewClusterID]), (k, currentNewClusterID)))

            currentNewClusterID += 1

        currentIteration += 1
```

Rysunek 6: Hierarchiczna metoda klasteryzacji

Mając daną historię połączeń klastrów możemy w prosty sposób uzyskać klasteryzacje dla wszystkich wartości k w czasie $\mathcal{O}(n)$ korzystając ze struktury DSU.

Warto napomnieć, że w przeciwieństwie do poprzedniego algorytmu, nie występuje tutaj losowość. To oznacza, że dla danych punktów i metryki na klastrach, dendrogram będzie zawsze wyglądał tak samo.

```python
def GetClusterizationFromHistory(connectionHistory, desiredClusterNumber):
    n = len(connectionHistory) + 1 #number of points

    D = DSU(2*n)

    for i in range(n - desiredClusterNumber):
        D.union(connectionHistory[i][0][0], connectionHistory[i][0][1])
        D.union(connectionHistory[i][0][0], i+n)

    return [D.find(i) for i in range(n)]
```

Rysunek 7: Uzyskiwanie podziału dla zadanej docelowej wartości klastrów

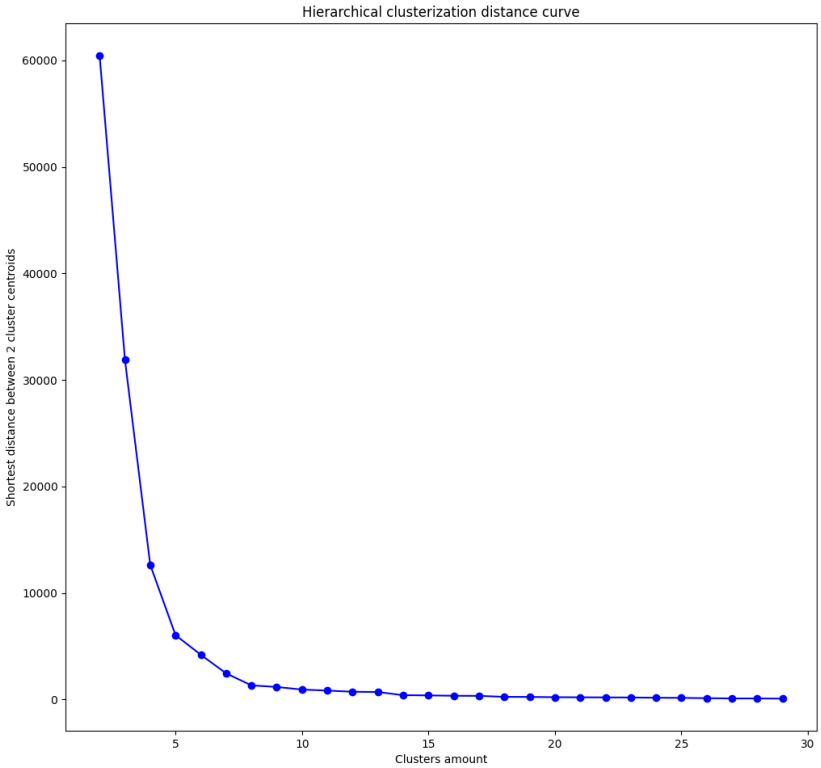
Pozostaje pytanie: Dla danego przebiegu algorytmu, jak określić, jaka ilość klastrów najlepiej opisuje zbiór danych wejściowych? Skorzystamy z podobnych metod co w pierwszym algorytmie. Podczas śledzenia procesu łączenia klastrów możemy zapamiętywać, jaką odległość miały 2 najbliższe klastry w momencie łączenia. Możemy następnie te odległości przedstawić na wykresie. Spodziewamy się, że gdy zaczniemy łączyć klastry znajdujące się wyraźnie daleko od siebie, to powinniśmy cofnąć się i zwrócić poprzednią liczbę klastrów. Analogicznie, szukamy tak zwanego punktu łokciowego

Oczywiście, możemy również skorzystać z uproszczonego współczynnika sylwetkowego, musimy tylko obliczyć w końcowej fazie centroidy uzyskanych klastrów.

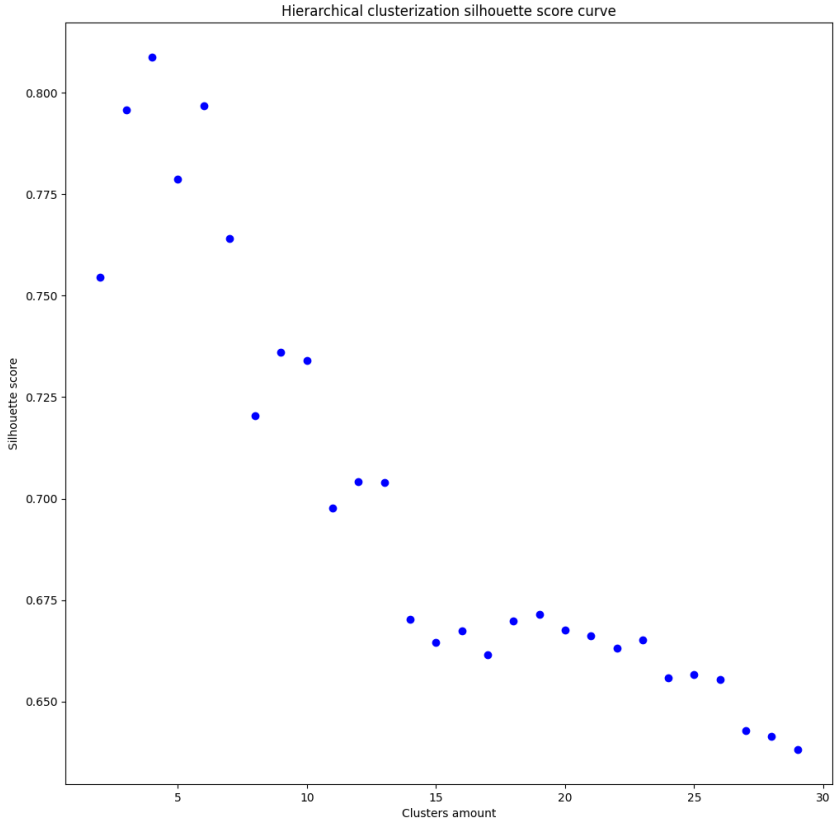
Na następnej stronie widoczne są wykresy zależności wspomnianych parametrów od wybranego miejsca ucięcia na dendrogramie, czyli współczynnika k.

Możemy wyciągnąć analogiczne wnioski do tych przedstawionych w algorytmie k-means. Na krzywej widzimy wyraźne wypłaszczenie, jednak nie mamy pewności, który punkty będzie najlepszym wyborem. Punkty 5, 6, 7 wyglądają równie dobrze. Przy pomocy parametru sylwetkowego możemy zobaczyć, że w punkcie $k = 6$ mamy lokalne ekstremum, zatem prawdopodobnie jest to najlepsza estymacja rzeczywistej liczby klastrów.

Przy analizie nowych danych bedziemy się więc posługiwać obydwoma współczynnikami.



(a) Wykres parametru najkrótszej odległości



(b) Wykres parametru sylwetkowego

1.3 Grupowanie spektralne

Poprzednio omówione algorytmy są przystosowane głównie do podziału na klastry o sferycznym kształcie, dużo gorsza skuteczność osiągają one na bardziej skomplikowanych, nieliniowych podziałach. Metoda ta w zasadzie nie stanowi samodzielnie algorytmu klasteryzacji, ale pozwala na mapowanie oryginalnych punktów do innych przestrzeni, w których łatwiej odnaleźć bardziej skomplikowane zależności, gdzie następnie wykorzystuje standardowy algorytm, najczęściej jest to k-means++.

Algorytm jest oparty o teorię grafów. Zbiór danych będziemy reprezentować jako graf nieskierowany ważony, w którym pojedyncze rekordy to wierzchołki. Macierz tego grafu będzie reprezentować podobieństwo danych punktów, czyli im większa waga krawędzi, tym bardziej punkty są do siebie podobne. Istnieje kilka najbardziej znanych sposobów konstrukcji grafu i odpowiadającej mu macierzy:

- Graf pełny, w którym waga krawędzi oparta jest o funkcję jądrową, najczęściej funkcję Gaussowską **RBF**. Dane punkty x_i, x_j łączymy krawędzią o wadze $\exp(-\gamma||x_i - x_j||_2)$, gdzie γ jest wybranym z góry parametrem
- Graf ϵ-bliskich sąsiadów, w którym punkty x_i, x_j łączymy krawędzią o wadze 1 wtedy i tylko wtedy, gdy $||x_i - x_j||_2 < \epsilon$ dla ustalonego ϵ
- Graf k-najbliższych sąsiadów, w którym krawędź pomiędzy x_i a x_j istnieje wtedy i tylko wtedy, gdy x_i jest wśród k najbliższych sąsiadów x_j, lub x_j jest wśród k najbliższych sąsiadów x_i.

Mając gotową macierz sąsiedztwa A tworzymy macierz diagonalną D, w której $D_{i,i} = \sum_{j=1}^{n} A_{i,j}$, czyli po prostu sumujemy wagi krawędzi wychodzących z każdego wierzchołka. Następnie tworzymy znormalizowaną macierz Laplaciana $L = D^{-\frac{1}{2}}(D - A)D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.

Dla macierzy L obliczamy jej k najmniejszych niezerowych wartości własnych (dla macierzy Laplaciana, 0 zawsze jest wartością własną dla wektorów stałych) oraz odpowiadające im wektory własne. Wektory tworzą macierz $W \in \mathbb{R}_{n \times k}$, którą traktujemy jako lista reprezentacji punktów z naszego oryginalnego zbioru danych w nowej, k-wymiarowej przestrzeni.

Na powstałej nowej przestrzeni punktów stosujemy już standardowy algorytm klasteryzacji, najczęściej będziemy korzystać z k-means, ale wypróbujemy również grupowanie hierarchiczne.

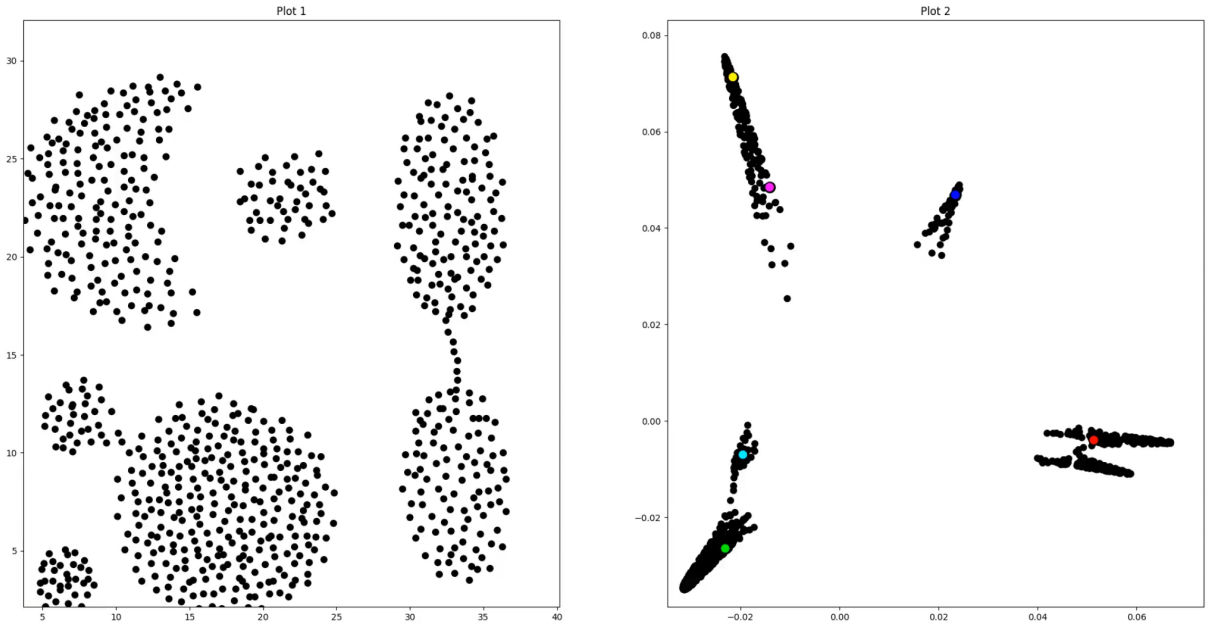
Liczbę klastrów wybieramy tak samo jak w algorytmie k-means. Nieco bardziej złożona jest optymalizacja parametrów używanych do budowania macierzy podobieństwa.

Dla grafu pełnego z funkcją RBF, chcemy, żeby parametr γ był proporcjonalny do typowej lokalnej odległości między punktami w zbiorze danych. Możemy zastosować taką heurystykę: dla każdego punktu obliczamy odległość do jego l-tego najbliższego sąsiada. Tutaj l zwykle jest proporcjonalne do $\sqrt{n}$. Następnie obliczamy średnią tych odległości dla wszystkich punktów i ustawiamy $\gamma = \frac{1}{2d_k^2}$, gdzie d_k jest wspomnianą średnią. Możemy również zbadać pewną skalę możliwych wartości parametru i porównywać wyniki, jednak jest to dosyć kosztowne obliczeniowo.

W przypadku grafu k-najbliższych sąsiadów, zazwyczaj interesuje nas parametr k równy lub nieco większy od przewidywanej liczby klastrów. Możemy więc zastosować tradycyjne algorytmy, a następnie dla znalezionego w nich optymalnego k wypróbować transformację spektralną.

```python
def ConstructAdjacencyMatrix(points, matrixConstructMethod, parameter):

    #parameter is:
    #   gamma for RBF
    #   epsilon for E_DIST
    #   k for KNN

    n = len(points)
    A = np.zeros((n,n))

    match matrixConstructMethod:
        case MatrixConstructMethod.FULL_RBF:
            for i in range(n):
                for j in range(n):
                    A[i][j] = math.exp(-parameter * utils.EuclideanDistSquared(points[i], points[j]))

        case MatrixConstructMethod.E_DISTANCE:
            for i in range(n):
                for j in range(n):
                    A[i][j] = 1 if utils.EuclideanDistSquared(points[i], points[j]) <= parameter else 0

        case MatrixConstructMethod.KNN:
            for i in range(n):
                closestPointsIndeces = np.argsort([utils.EuclideanDistSquared(points[i], points[j]) for j in range(n)])
                for j in range(1, parameter+1): #we skip i itself which is the closest
                    A[i][closestPointsIndeces[j]] = 1

        case _:
            raise ValueError("ConstructAdjacencyMatrix: invalid construct method!")

    return A
```

(a) Różne metody konstrukcji macierzy sąsiedztwa



(b) Przykład transformacji z użyciem funkcji RBF i $\gamma = 0.1$ danych z pierwszego zbioru

2 Ocena skuteczności algorytmów

Mając do wyboru wiele algorytmów klasteryzacji potrzebujemy umieć ocenić ich jakość. W przypadku danych, do których otrzymujemy poprawne przyporządkowanie klastrów, nie jest to szczególnie trudne. Możemy zliczać pary punktów, które we właściwym podziale występują lub nie występują w jednym klastrze, a nasz algorytm twierdzi na odwrót. Symbolicznie, dla zbioru danych $X = \{x_1, x_2, ..., x_n\}$, danej właściwej funkcji klasteryzacji $p : X \to \mathbb{N}$ oraz funkcji zwróconej przez algorytm $q : X \to \mathbb{N}$:

$$J(q) = \frac{1}{\binom{n}{2}} \sum_{i,j \in [n], i<j} \mathbf{1}[p(i) = p(j) \iff q(i) = q(j)]$$

Nie jest to może idealna miara w takim znaczeniu, że próg dobrej wartości może się znacznie różnić dla poszczególnych zbiorów danych, jednak sprawdza się ona przy porównywaniu kilku algorytmów na jednym, takim samym zbiorze danych.

Jeśli dane są np. punktami na płaszczyźnie, tak jak w przypadku wielu plików badanych w projekcie, możemy czytelnie przedstawić dane na wykresie wraz z przypisanymi przez nas klastrami i ocenić, czy algorytm był skuteczny.

Czasami mamy jednak do czynienia z danymi bez przypisanych klas, które trudno zwiualizować, tak jak np. w przypadku danych z pliku *dane_18D.txt*. Możemy w takiej sytuacji próbować zredukować wymiar tak, aby można było dane przedstawić czytelnie na wykresie. Opieramy się również o wyniki numeryczne przedstawione w poprzednim dziale, takie jak współczynnik sylwetkowy czy metoda łokcia.

```python
def CalculateClusterizationScore(realClasses, computedClasses):

    c1 = NormalizeClassification(realClasses)
    c2 = NormalizeClassification(computedClasses)

    n = len(realClasses)

    return (1/(n*(n-1))) * sum(1 for i in range(n) for j in  range(n) if
    ((i != j) and
     ((realClasses[i]==realClasses[j]),(computedClasses[i]==computedClasses[j])) in [(True, True), (False, False)]))
```

Rysunek 10: Obliczanie wskaźnika jakości algorytmu klasteryzacji

3 Analiza danych

3.1 Pliki dane_2D_n.txt - podstawy

Na początku zajmiemy się danymi ze wspomnianych plików. Ponieważ reprezentują one punkty na płaszczyźnie, możemy ładnie zwizualizować przebiegi obydwu algroytmów, zarówno na oryginalnych punktach jak i na transformacjach do innych przestrzeni dwuwymiarowych. Wideo zawierające przebiegi poszczególnych algorytmów umieszczę w katalogu Miniprojekt4/videos.

Dla każdego zbioru wykorzystamy omówione wcześniej metody dla algorytmów kmeans oraz grupowania hierarchicznego, żeby wyznaczyć optymalną liczbę klastrów.

Wartości parametrów będziemy porównywać dla $k \in [2, \sqrt{n}]$. Być może pierwiastek z liczby rekordów to trochę zbyt łagodne ograniczenie górne, ale lepiej przedobrzyć w tę stronę, niż w drugą, a czas obliczania parametrów i tak był dość krótki.

Wyniki dla algorytmu k-means będą przedstawione jako średnie wyników uzyskanych podczas kilkunastu iteracji. W grupowaniu hierarchicznym wystarczy nam jeden przebieg, ponieważ algorytm jest deterministyczny.

Na kolejnych stronach przedstawione są grupy wykresu. Na każdej stronie widnieje rozkład punktów w badanym zbiorze oraz 4 wykresy, ilustrujące kolejno wartości współczynnika MSE i sylwetkowego w algorytmie kmeans oraz odległość najbliższych klastrów i współczynnik sylwetkowy algorytmu hierarchicznego. Po uzyskaniu wyników możemy dla każdego ze zbiorów wyznaczyć przewidywaną liczbę klastrów. Strona bezpośrednio po obecnej przedstawia podsumowanie wniosków wyciągniętych na podstawie analizy uzyskanych wyników, oraz wybrane liczby klastrów, które będą testowane dla tych dwóch algorytmów.

Następnie przedstawię uzyskane klasteryzacje oraz ich wyniki. Zmienna *Score* na umieszczonych niżej wykresach oznacza wartość funkcji J dla konkretnego klasyfikatora.

Przedstawię słowne podsumowanie uzyskanych rezultatów:

- Algorytmy znakomicie poradziły sobie z klasteryzacją zbiorów 5 i 6. W obydwu przypadkach nieco lepsze wyniki uzyskaliśmy przy pomocy grupowania hierarchicznego, były one niemalże identyczne z rzeczywistymi, różniły się jedynie w punktach będących szumem, które nie sposób poprawnie sklasyfikować bez dodatkowej wiedzy o nich. Jak już wielokrotnie wspominaliśmy, klastry o kształtach sferycznych to ich specjalność
- W przypadku niektórych zbiorów nie zostały wyłapane niektóre subtelności. Zadziwiające są uzyskane klasyfikacje dla zbioru 1, gdzie obydwa algorytmy pomyliły się przy klasteryzacji dwóch małych klastrów po lewej. Klasyfikator hierarchiczny bardzo dobrze podzielił zbiór na 6 części uzyskując wartość funkcji J na poziomie 99.5 ale, jak widać, ostatnim krokiem nie było połączenie dwóch wspomnianych zbiorków. Podobnie w przypadku zbioru 2, gdzie klaster znajdujący się całkowicie wewnątrz innego klastru był w zasadzie niewykrywalny. To jednak było dość spodziewanym rezultatem, spróbujemy tutaj skorzystać z klasteryzacji spektralnej. Problematyczne było również oddzielenie mocno oddalonych punktów z prawej strony od większej wysepki na ich środku.
- Zbiory 7 i 8 doskonale pokazują, że nawet gdy mamy do czynienia z mocno skondensowanymi klastrami, to i tak ich nienaturalny, nie kulisty kształt prowadzi do nie najlepszych klasyfikacji ze strony badanych algorytmów. Dla zbioru 8 widać, że dla wyższych wartości k klasyfikacje przebiegały nieźle, jednak w ostatnim kroku algorytmy nie zdecydowały się na połączenie dwóch fragmentów rogalika, tylko łączyły jeden fragment ze sferą na górze. Mocno odstające od reszty punkty nie miały szczególnego wpływu na ich działanie, ponieważ zbiory były dostatecznie duże a klastrów dostatecznie niewiele.
- Struktury w pozostałych zbiorach właściwie nie zostały praktycznie wcale wyłapane. Widzimy, jak bardzo algorytmy te mogą się mylić, nawet, gdy podamy im liczbę klastrów zgadzającą się z rzeczywistą liczbą. Tutaj również spodziewamy się lepszych wyników korzystając z technik mapowania

3.1.1 Dane_2D_1

W przypadku obydwu algorytmów wykresy w lewej kolumnie ulegają wyraźnemu wypłaszczeniu dla $k = 6$. Wartości współczynnika sylwetkowego osiągają w nim wysokie wartości, po których następuje silny spadek, co wzmacnia przekonanie o tym, że ten parametr jest dobrym wyborem. Patrząc na rysunek widzimy, że klastrów prawdopodobnie jest 6 lub 7, nie mamy pewności, czy górna mała kropka jest połączona z największym klastrem na dole. Sklasyfikujemy więc zbiory dla $k \in [6, 7]$.

3.1.2 Dane_2D_2

Obydwa algorytmy notują niezwykle szybki spadek parametrów z lewej kolumny, możnaby z nich wyczytać, że prawdopodobną liczbą klastrów jest 4. Patrząc na rysunek możemy się jednak spodziewać, że obydwa algorytmy zaniżą rzeczywistą liczbę klastrów. Są one raczej dostosowane do wykrywania grup o kształcie sferycznym, mogą więc mieć duży problem z oddzieleniem środka pierścienia w lewym dolnym rogu od samego brzeg, albo z dopasowaniem pozornie losowych punktów z prawej strony do jednej grupy. W rzeczywistości jest ich pewnie 5 lub 6. Możemy z ciekawości zobaczyć, jakie klasyfikacje proponują algorytmy dla $k \in [3, 4]$, ale najprawdopodobniej lepsze wyniki uzyskamy stosując klasteryzację spektralną.

3.1.3 Dane_2D_3

Wyniki obydwu algorytmów wskazują na 3 lub 4 klastry, grupowanie hierarchiczne wydaje się szacować nieco większą średnią wartość niż kmeans. Patrząc na rysunek podejrzewamy, że $k = 3$ jest rzeczywistą liczbą klastrów, jednak algorytmy będą mieć problemy z dopasowaniem całej otoczki jako jeden klaster. Tutaj również zastosujemy grupowanie spektralne z oczekiwaniem dużo lepszych wyników.

3.1.4 Dane_2D_4

Obydwa algorytmy podają bardzo chaotyczne informacje dotyczące przewidywanej liczby klas w tym zbiorze. Z jednej strony, wśród niskich, spodziewanych przez nas, liczb klastrów, w $k = 3$ lub $k = 4$ widzimy wyraźne wypłaszczenia i skoki funkcji sylwetkowej. Funkcja ta uzyskuje jednak bardzo dobre wartości dla zadizwiająco dużych liczb potencjalnych klastrów. Dane tworzą kształty spirali, gołym okiem widzimy, że liczba klastrów prawdopodobnie wynosi 3. Sprawdzimy zarówno tę wartość jak i propozycje algorytmów co do większych liczb $k \approx 15$.

3.1.5 Dane_2D_5

Ten zbiór stanowi doskonały przykład wartości korzystania ze wskaźnika sylwetkowego. Patrząc jedynie na spadek wartości MSE lub odległości najbliższych klastrów, obydwa algorytmy prawdopodobnie zdecydowałyby się na zbyt niską wartość $k \approx 10$. Jest to największy zbiór danych (zawiera ok. 3000 rekordów), a ponadto, klastry są w kształcie sferycznym, co stanowi idealną sytuację dla obliczenia wskaźnika sylwetkowego. Widzimy, że w przypadku obydwu algorytmów, największą wartość osiąga on dla $k = 30$. W rzeczywistości wygląda na to, że klastrów jest 31, prawdopodobnie algorymy połączyły 2 bardzo blisko leżące grupy w jedną. Nie mniej jednak oba okazały się tutaj bardzo skuteczne.

3.1.6 Dane_2D_6

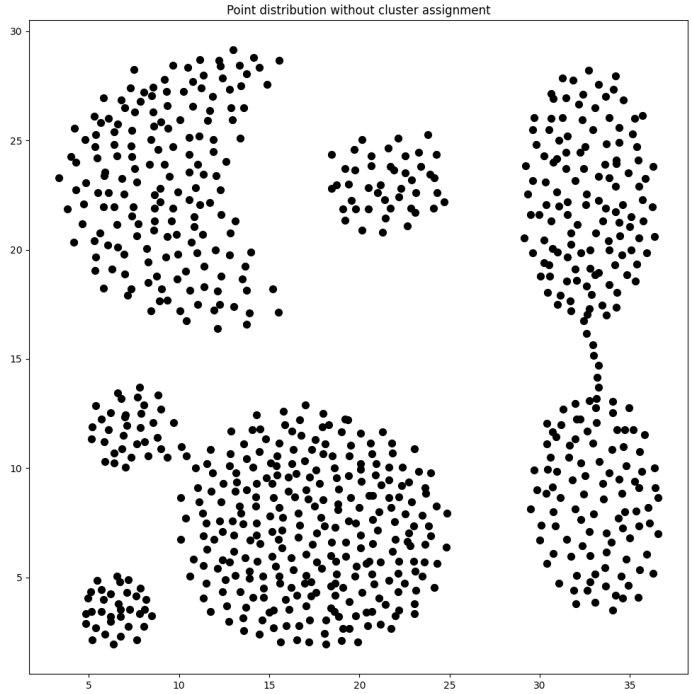
W tym przypadku wnioski są bardzo podobne jak w poprzednim. Wskaźnik MSE znów nie daje nam wielu informacji, dystansowy w grupowaniu hierarchicznym co prawda notuje ostatni wyraźny spadek w okolicy rzeczywistej liczby klastrów, ale nie jest do końca jasne, czy powinniśmy go uwzględniać. Wskaźniki sylwetkowe idealnie wskazują ektremum w $k = 15$, co wygląda na rzeczywistą liczbę klastrów.

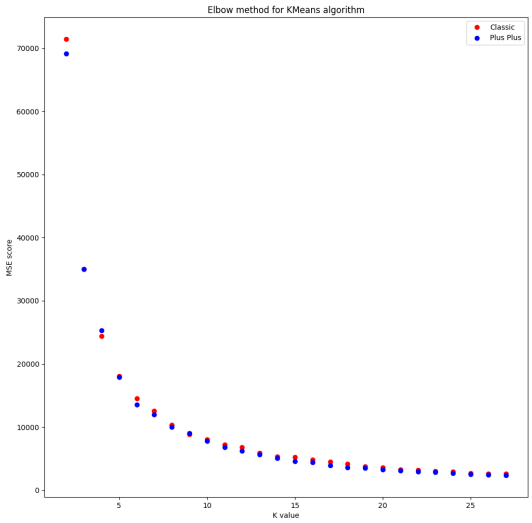
3.1.7 Dane_2D_7

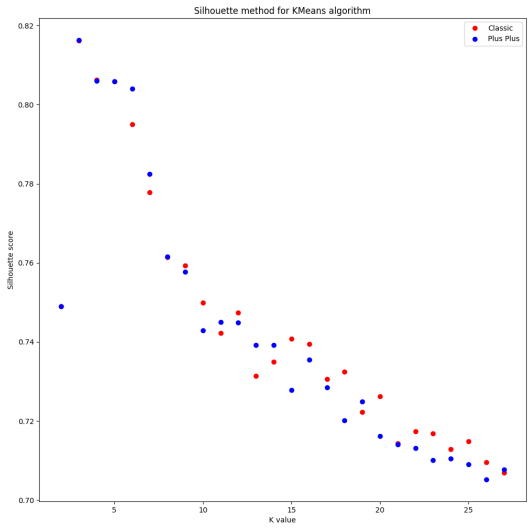
Kolejny zbiór danych dość trudny do rozpoznania przez oba podstawowe algorytmy. Obydwa wskazały $k = 8$ jako proponowaną liczbę klastrów, jednak wcale nie widać tego na rysunku. Podejrzewamy, że klastry będą 2, ale zobaczymy, jaki podział proponują algorytmy w przypadku wskazywanej przez nich liczby klastrów.

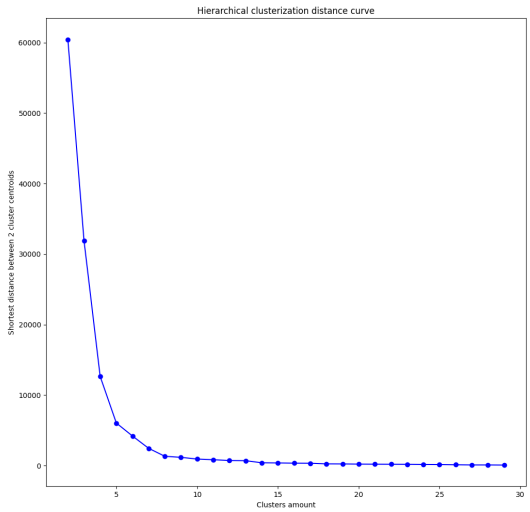
3.1.8 Dane_2D_8

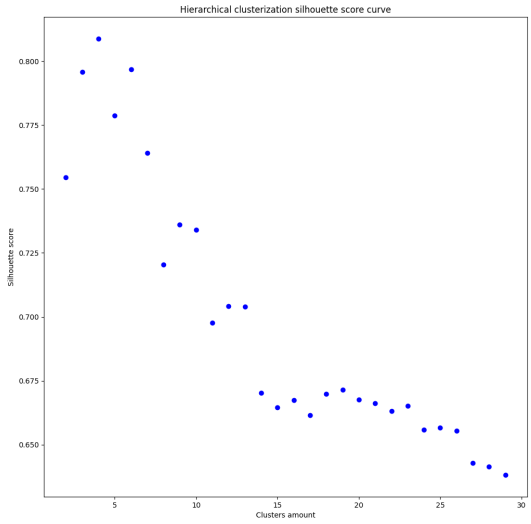
Algorytm k-means zdecydowanie wskazał wartość $k = 4$, hierarchiczny natomiast ma spore wątpliwości. Wydaje się, że wskazuje $k = 3$ ze względu na współczynnik sylwetkowy, jednak notuje on mocny spadek wskaźnika dopiero po tej wartości. W rzeczywistości sytuacja wygląda tak, jakby klastry były 2, a wysoką liczbę przewidywanych klastrów powodowało kilka odstających punktów.

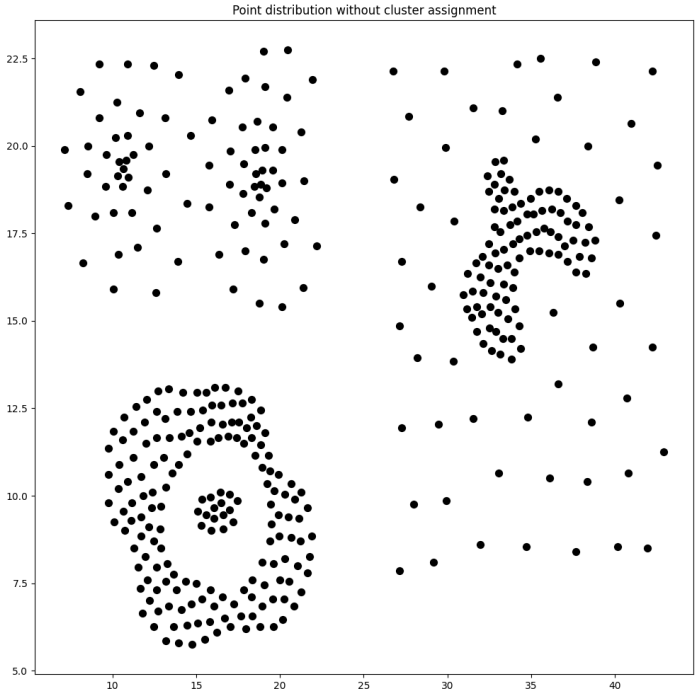

Point distribution without cluster assignment

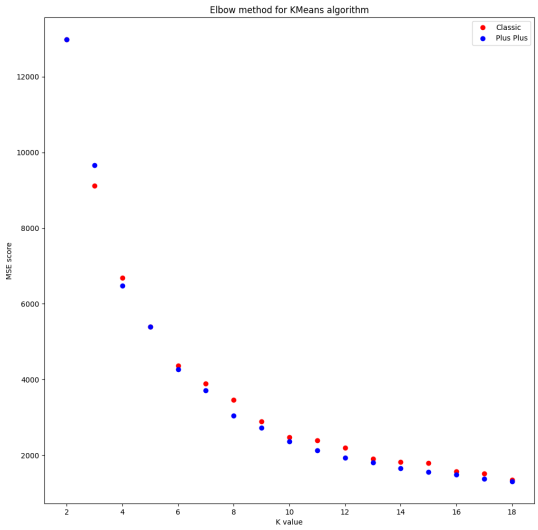

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

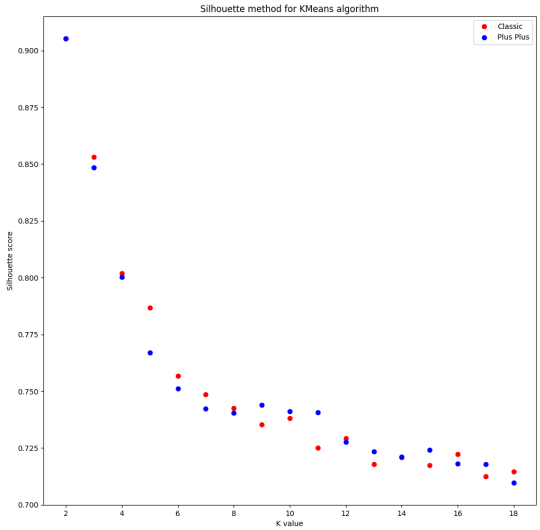

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

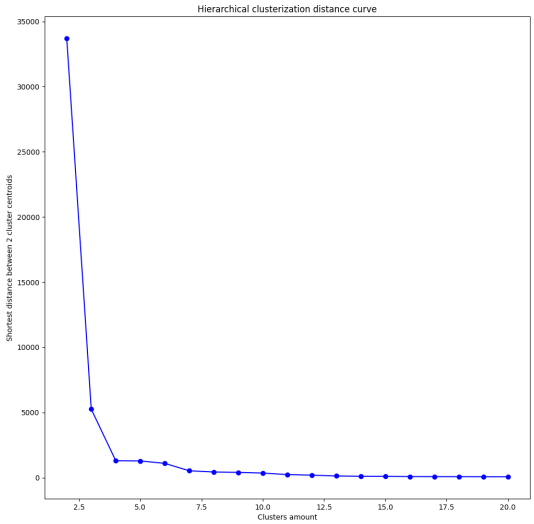

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

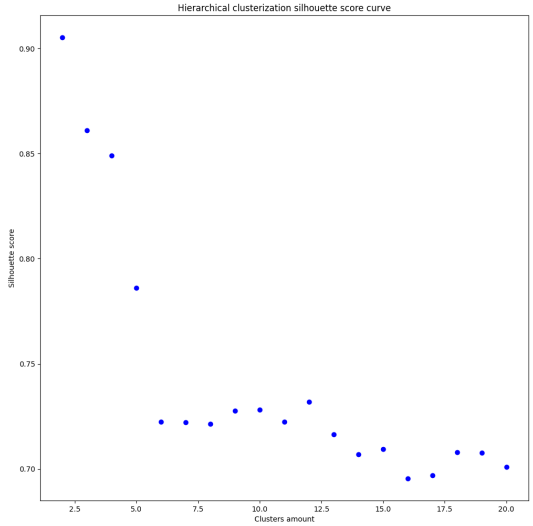

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

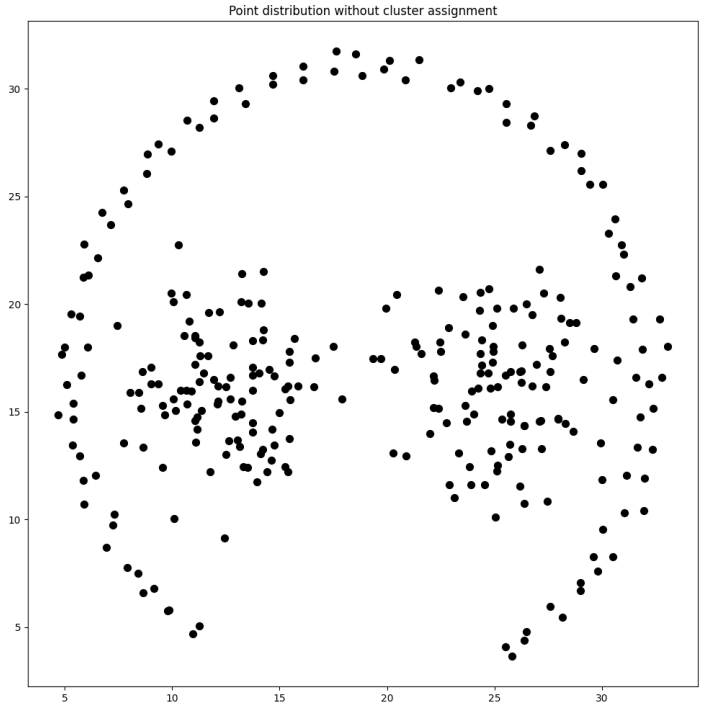

Point distribution without cluster assignment

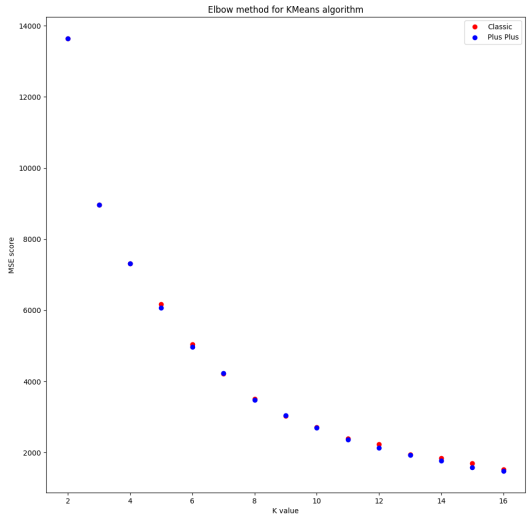

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

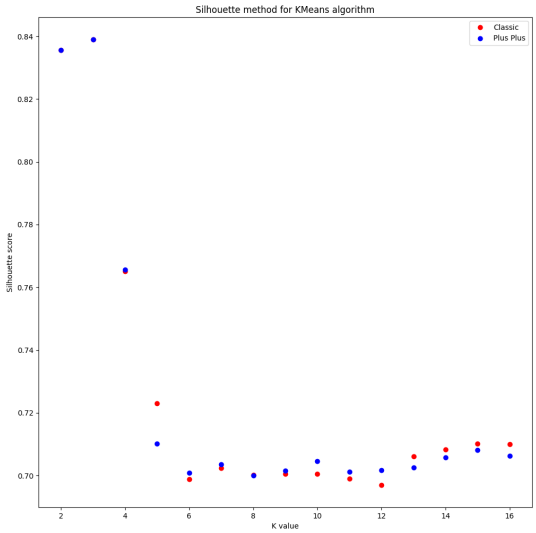

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

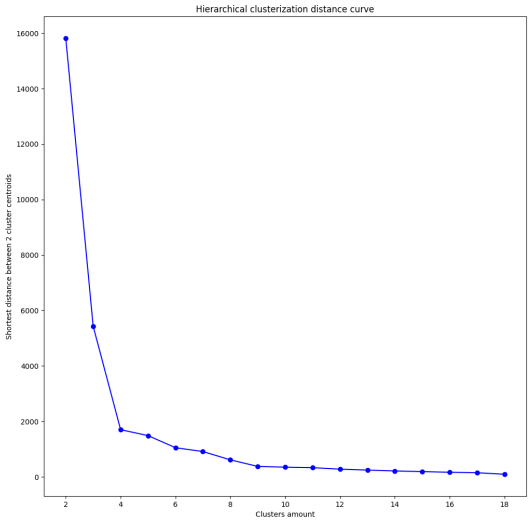

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

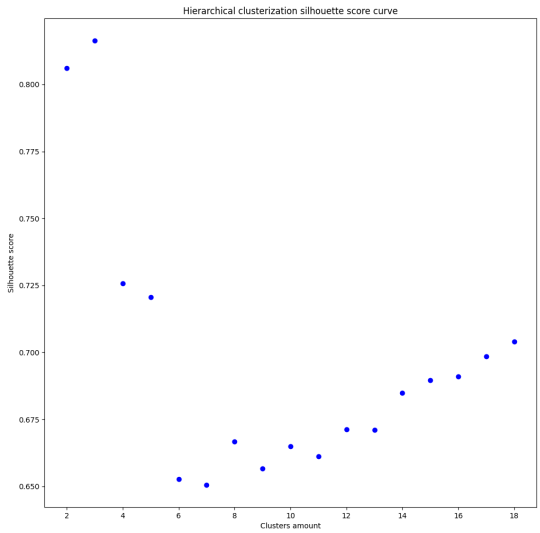

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

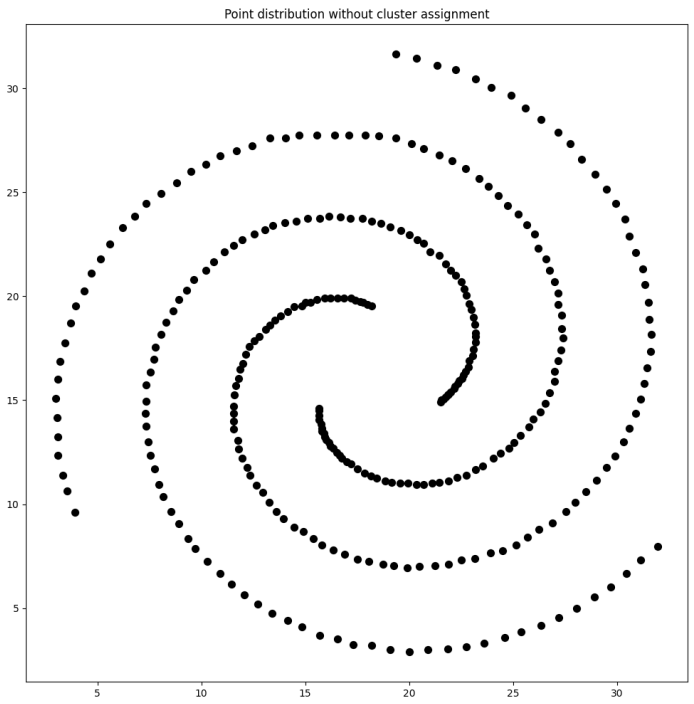

Point distribution without cluster assignment

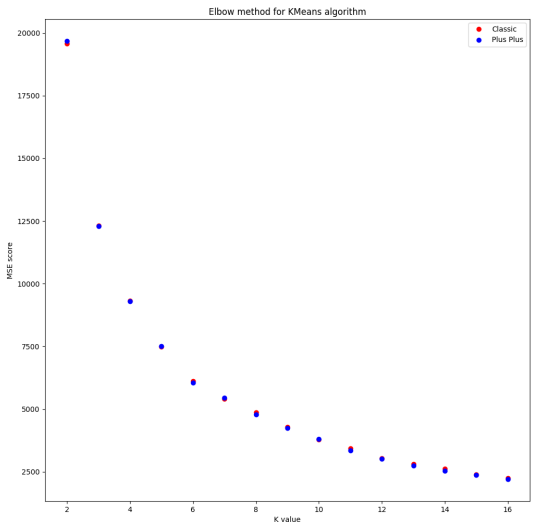

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

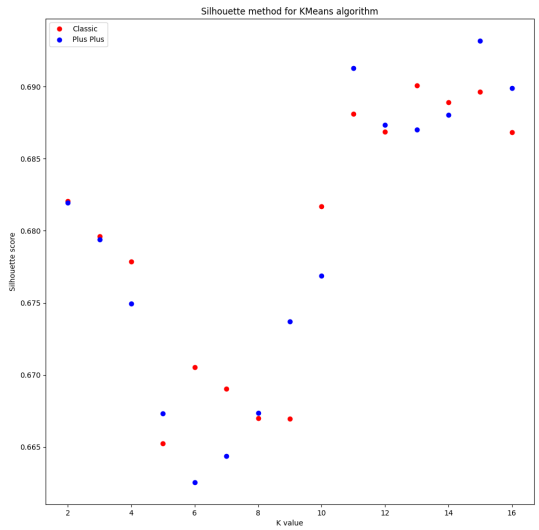

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

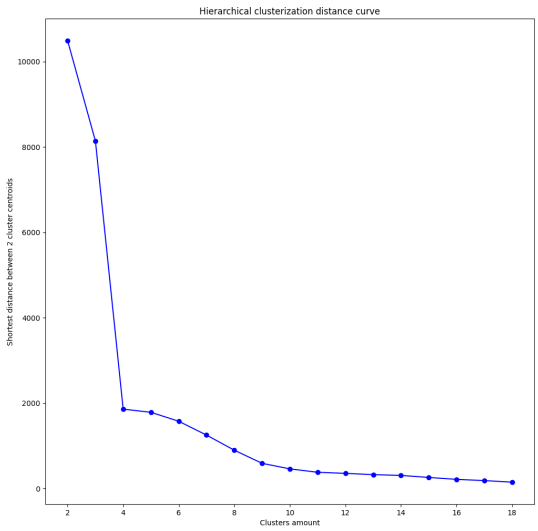

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

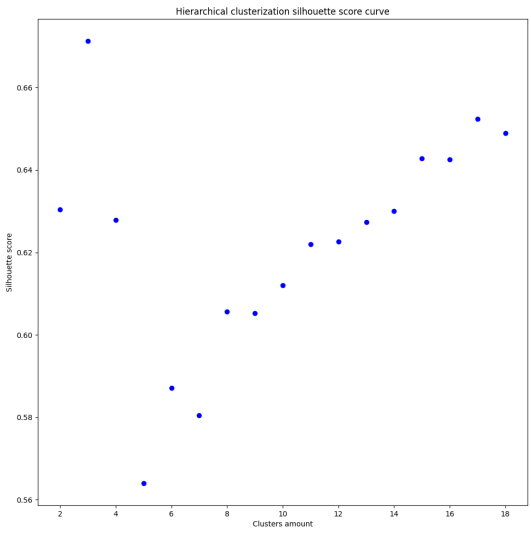

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

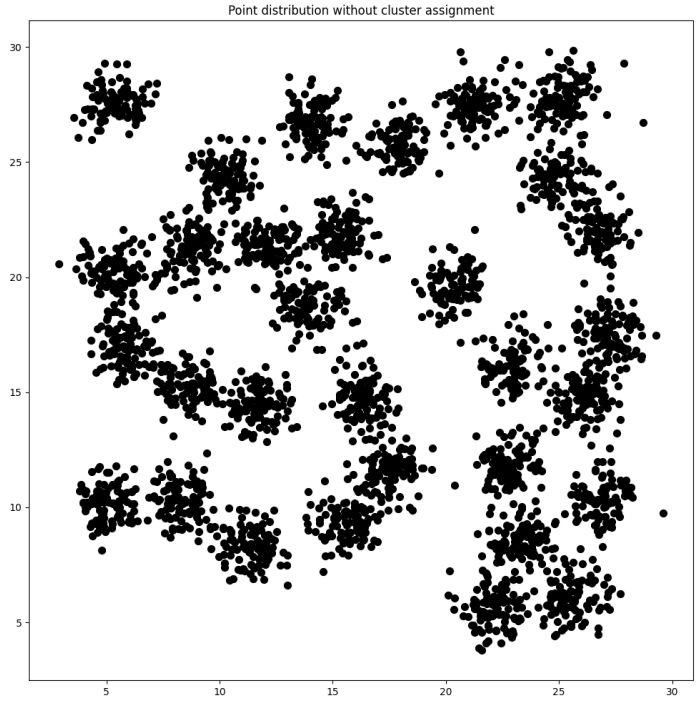

Point distribution without cluster assignment

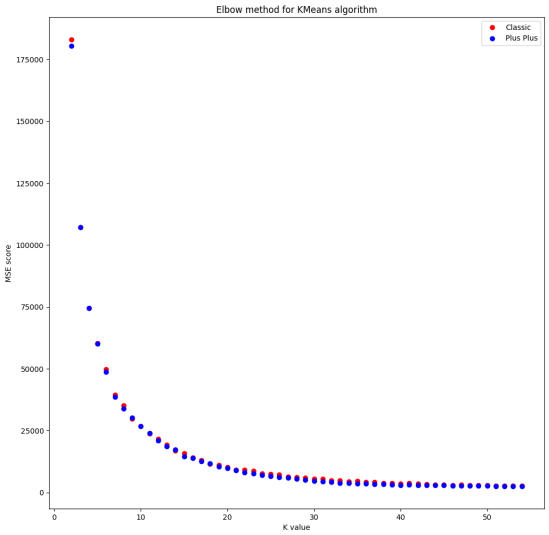

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

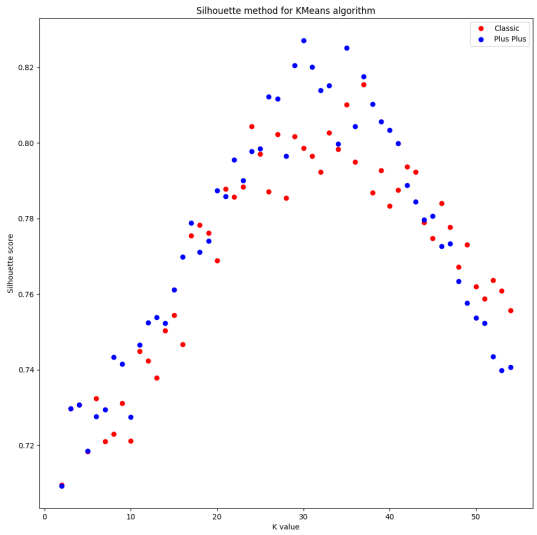

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

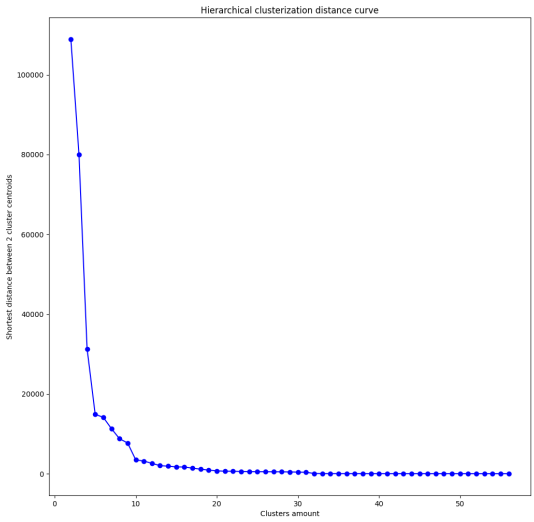

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

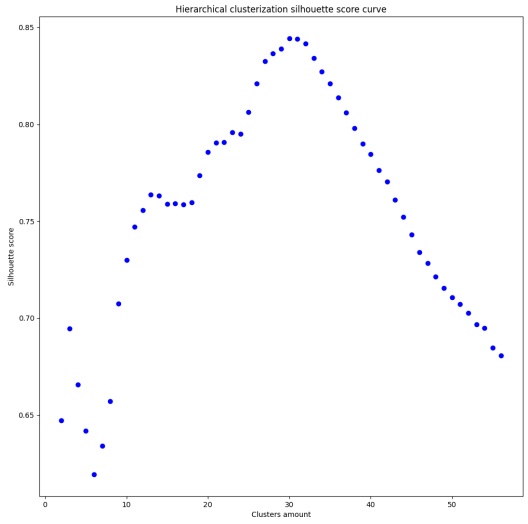

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

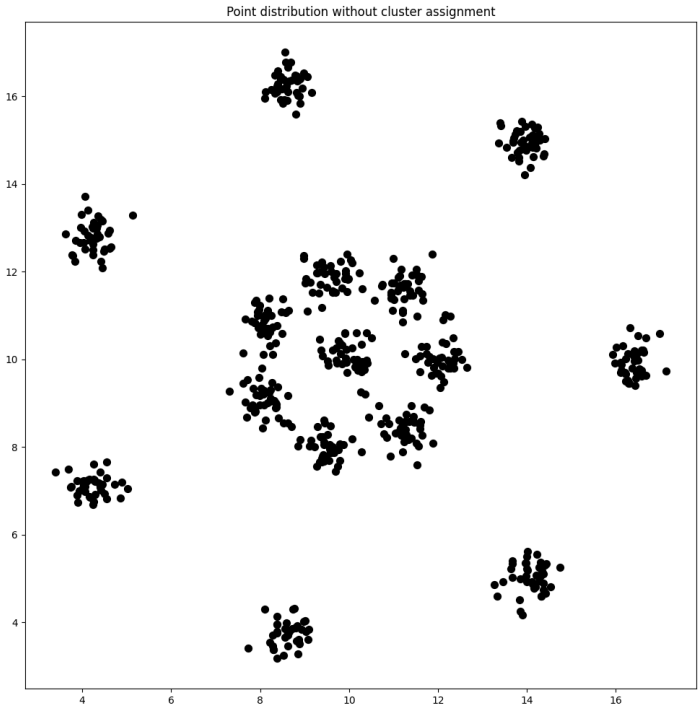

Point distribution without cluster assignment

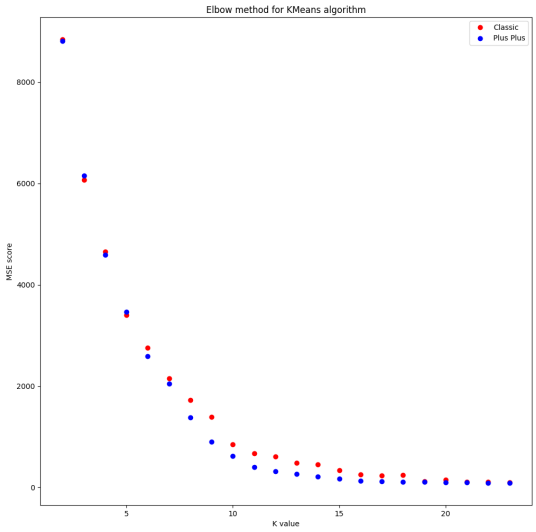

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

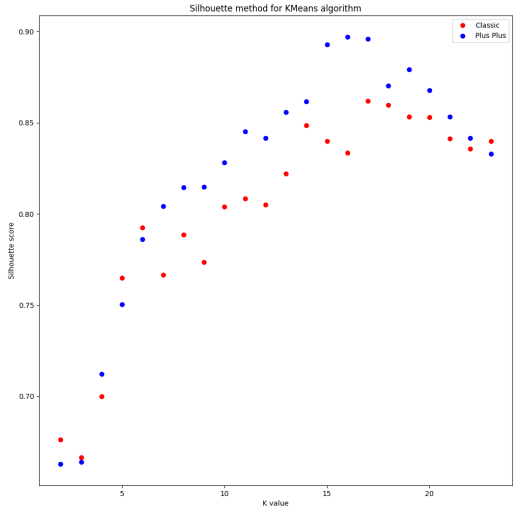

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

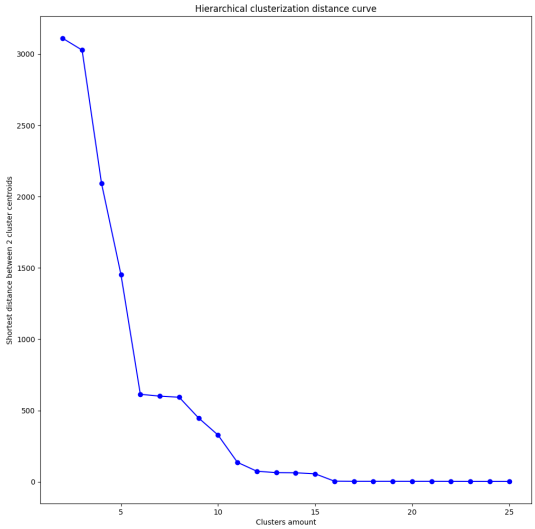

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

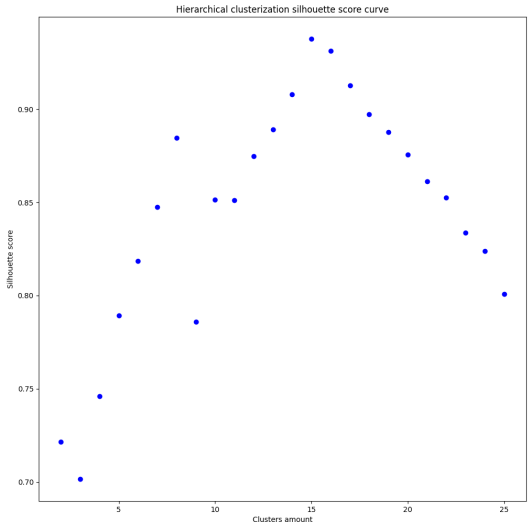

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

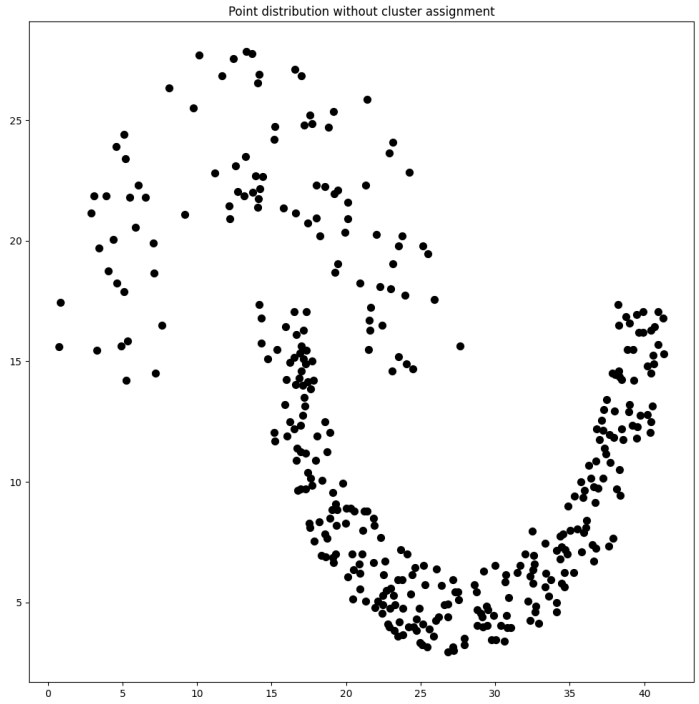

Point distribution without cluster assignment

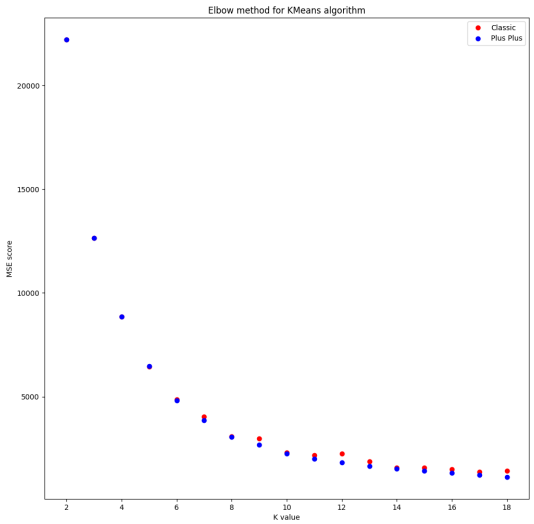

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

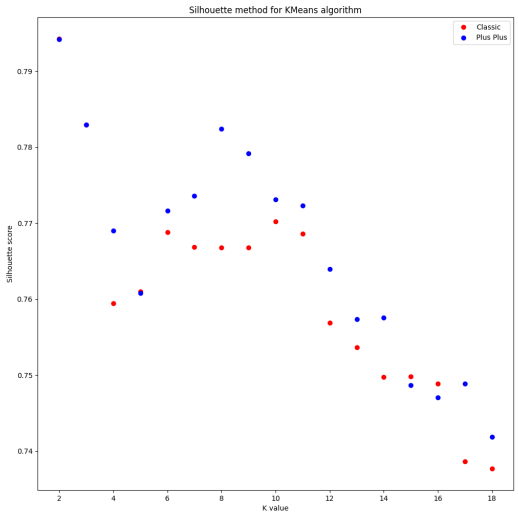

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

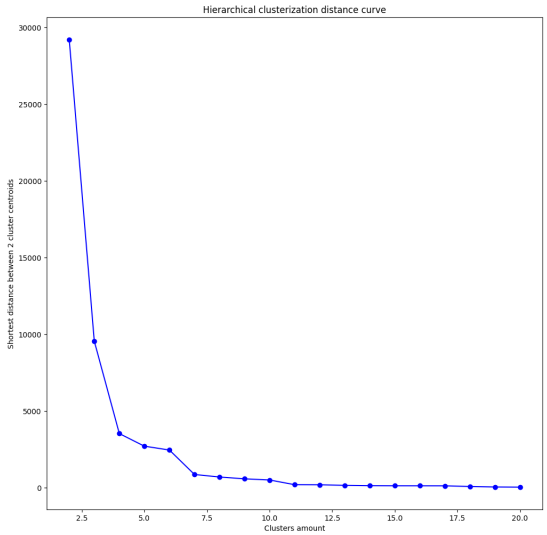

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

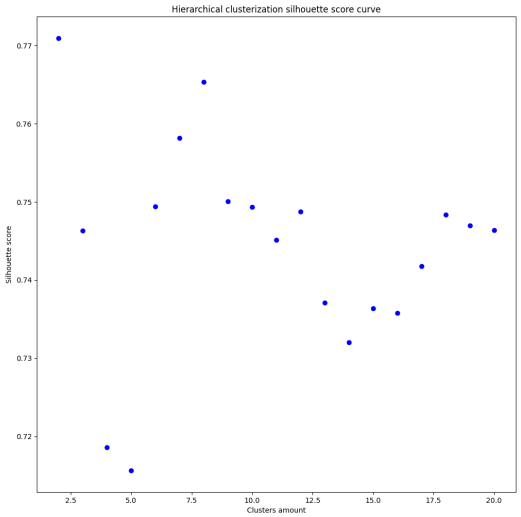

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

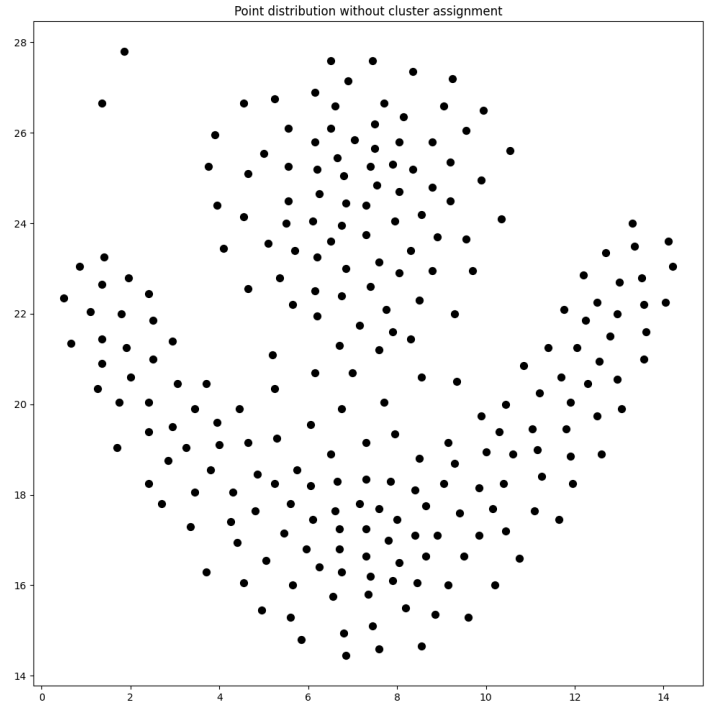

Point distribution without cluster assignment

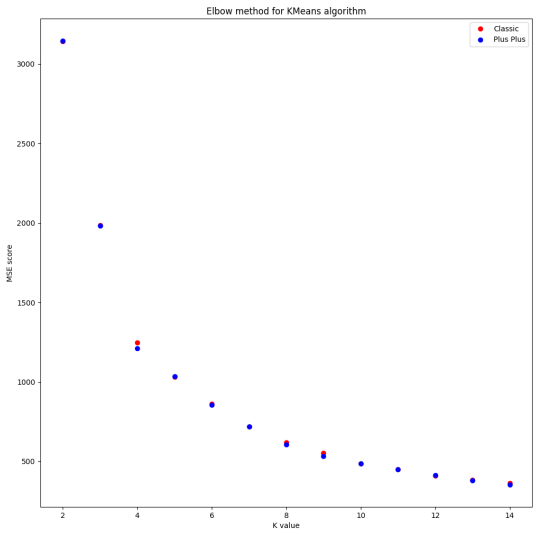

Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

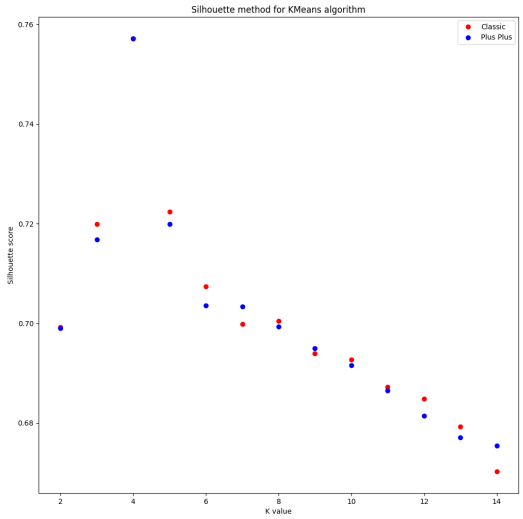

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value

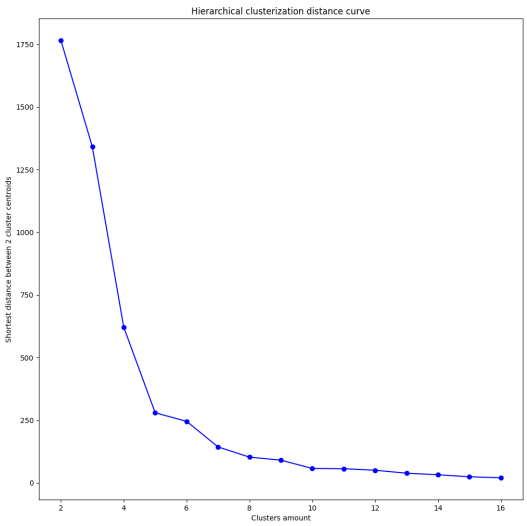

Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

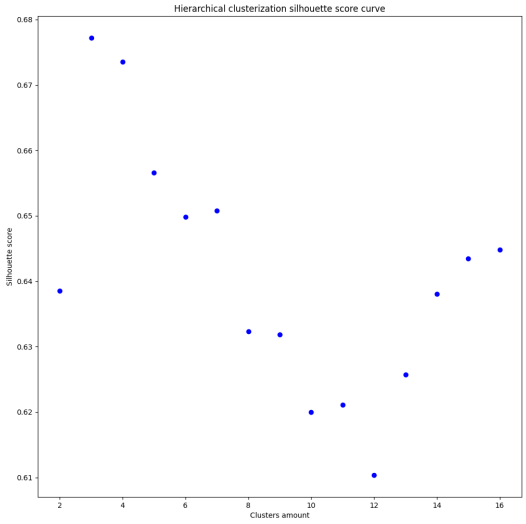

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount

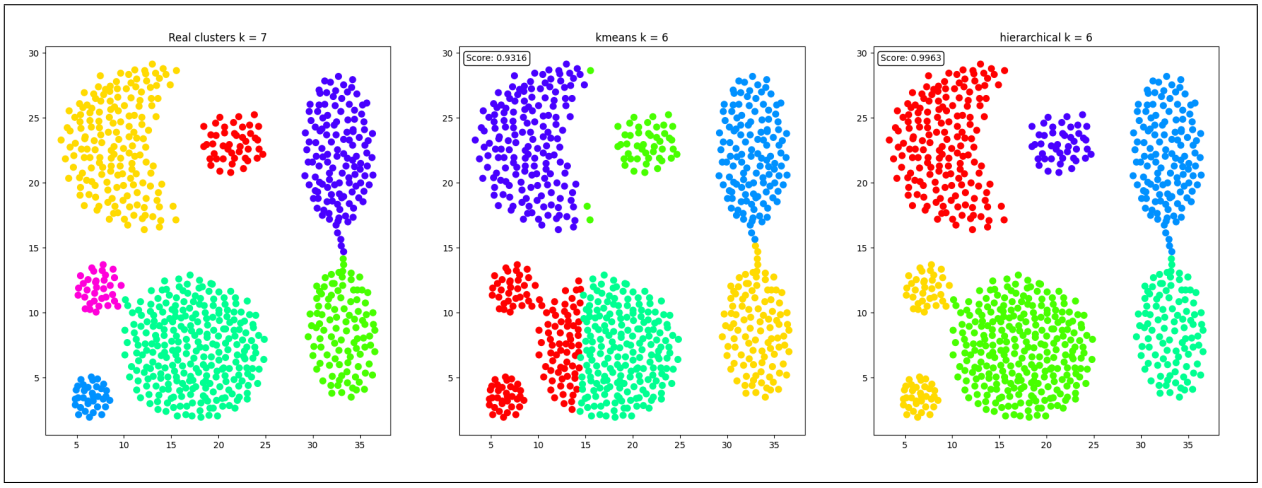

Point distribution without cluster assignment


Elbow method for KMeans algorithm
Classic
Plus Plus
MSE score
K value

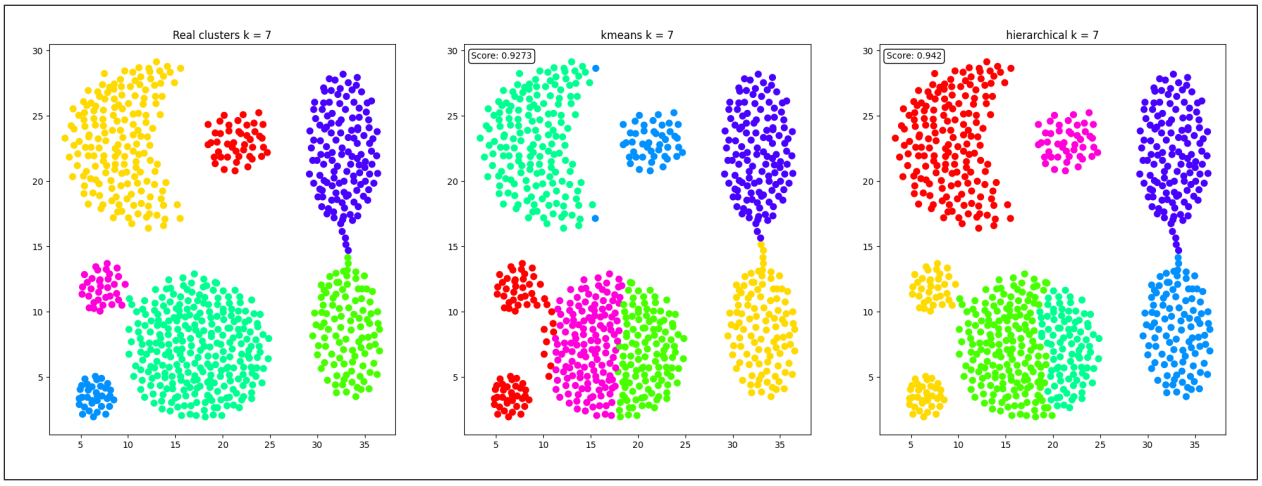

Silhouette method for KMeans algorithm
Classic
Plus Plus
Silhouette score
K value


Hierarchical clusterization distance curve
Shortest distance between 2 cluster centroids
Clusters amount

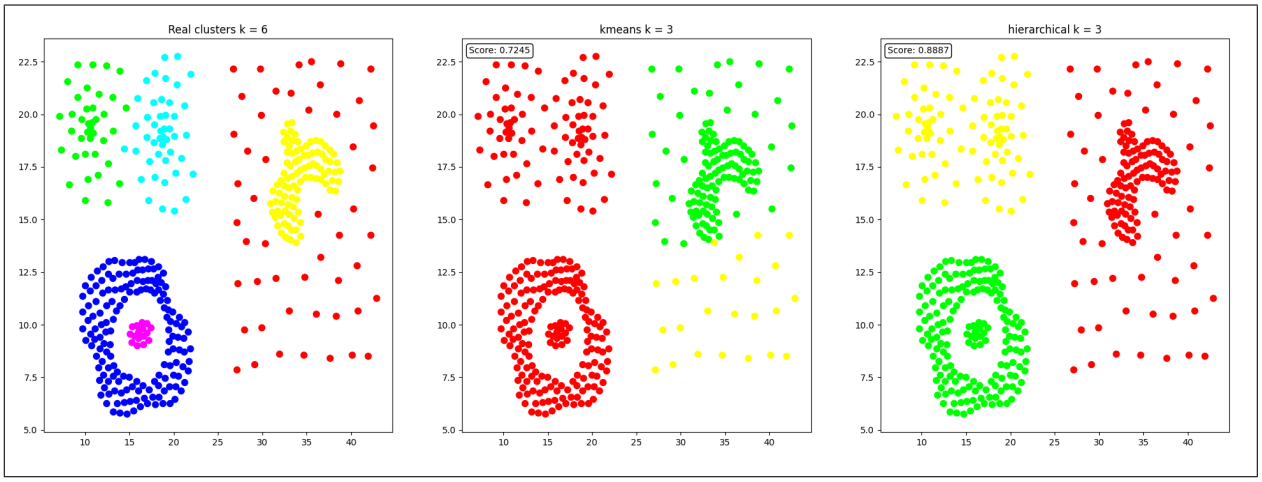

Hierarchical clusterization silhouette score curve
Silhouette score
Clusters amount



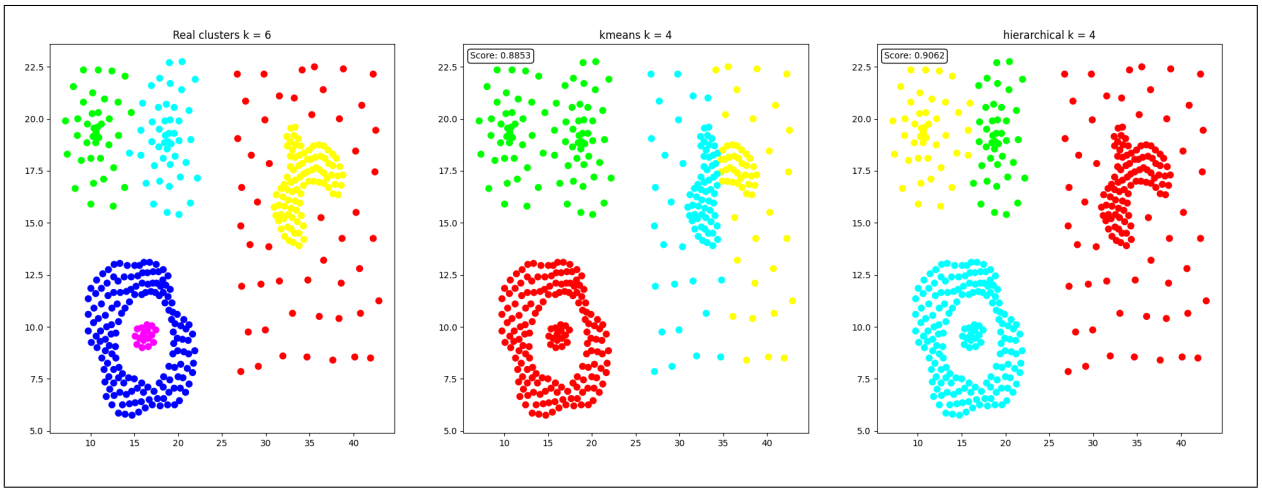
Rysunek 19: Dane 1, $k = 6$



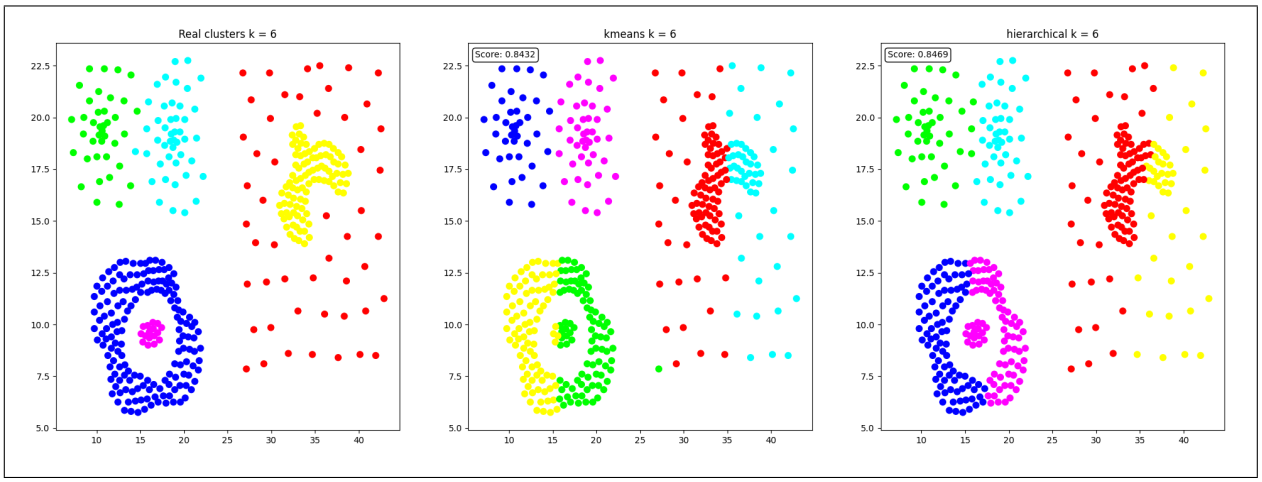
Rysunek 20: Dane 1, $k = 7$



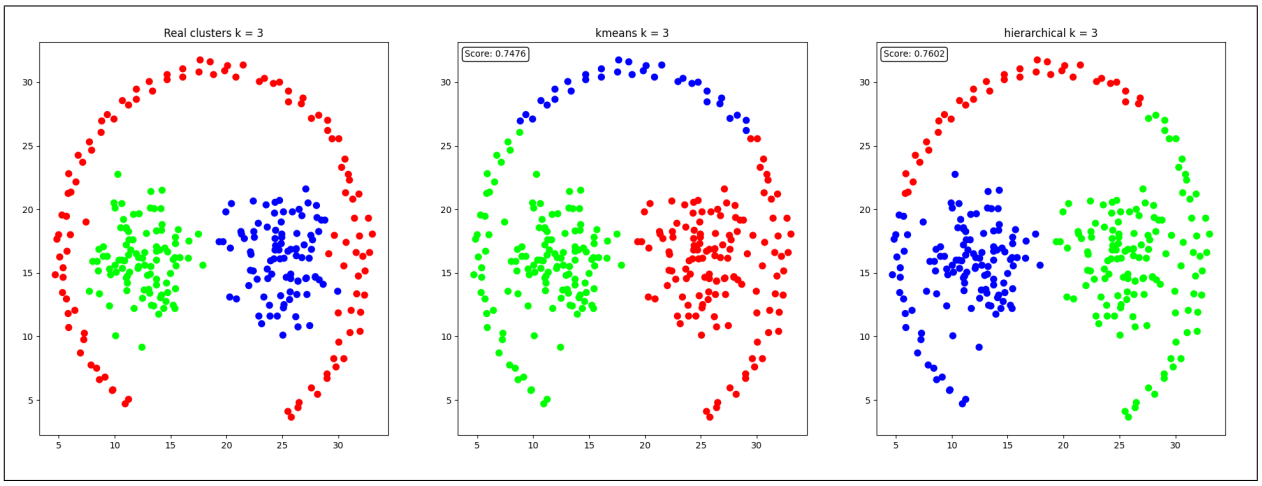
Rysunek 21: Dane 2, $k = 3$



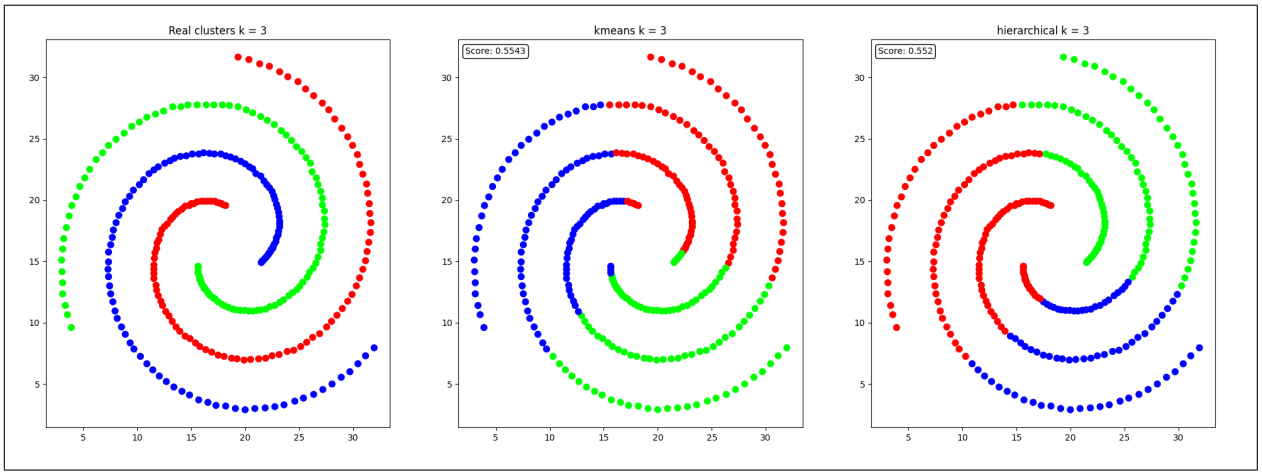
Rysunek 22: Dane 2, $k = 4$



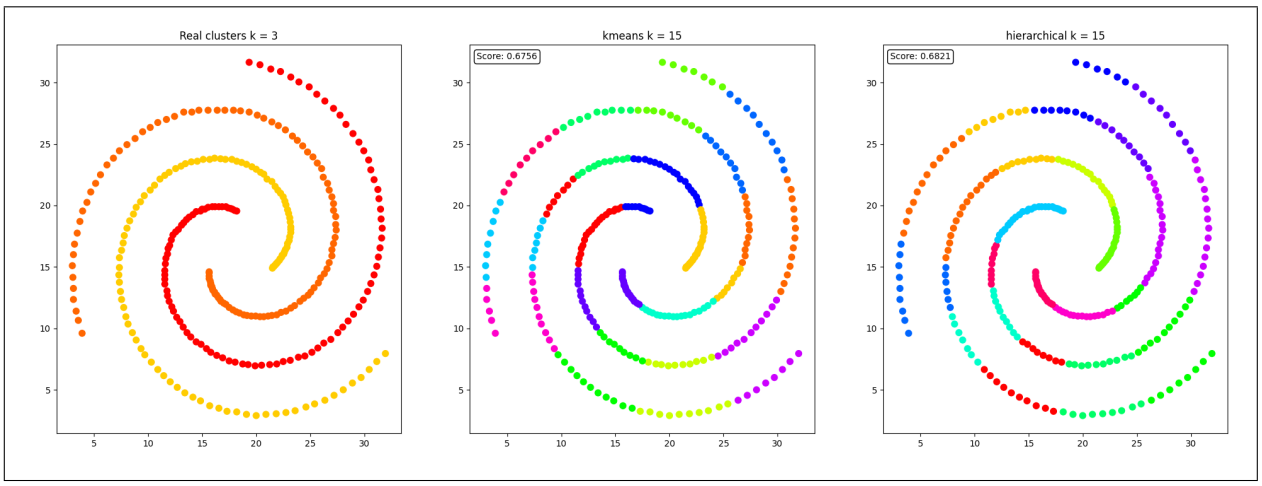
Rysunek 23: Dane 2, $k = 6$



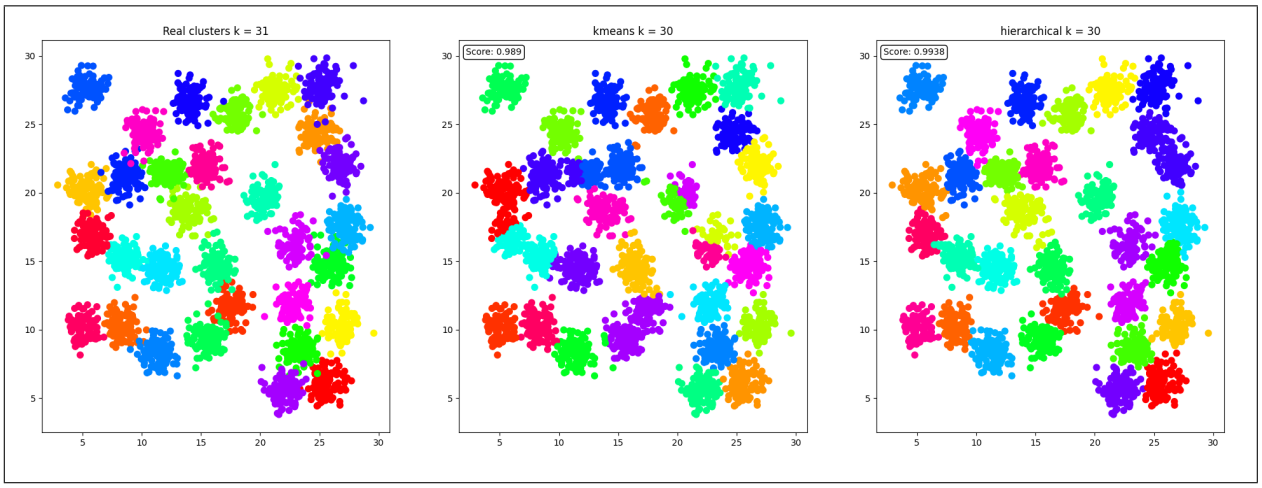
Rysunek 24: Dane 3, $k = 3$



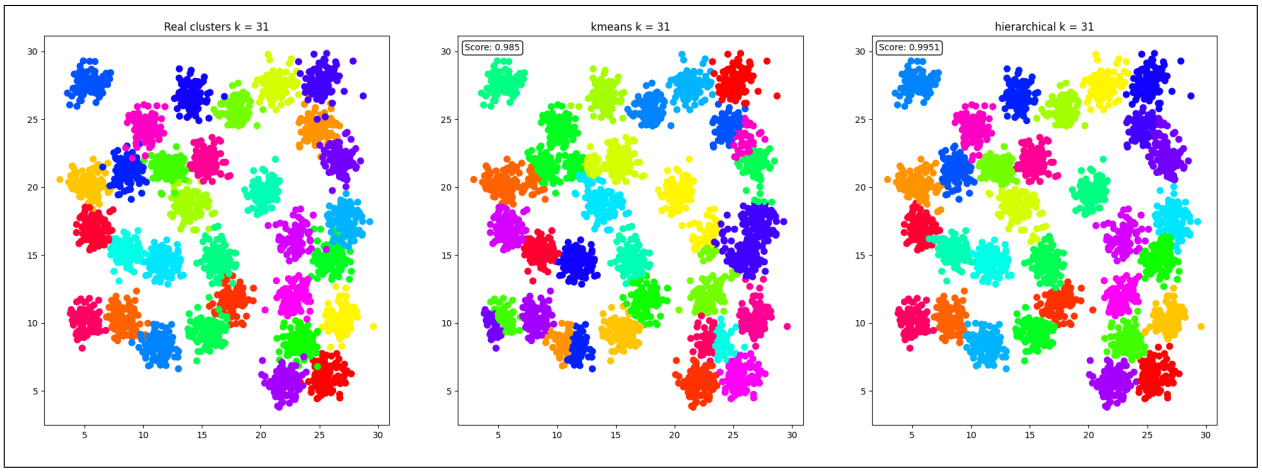
Rysunek 25: Dane 4, $k = 3$



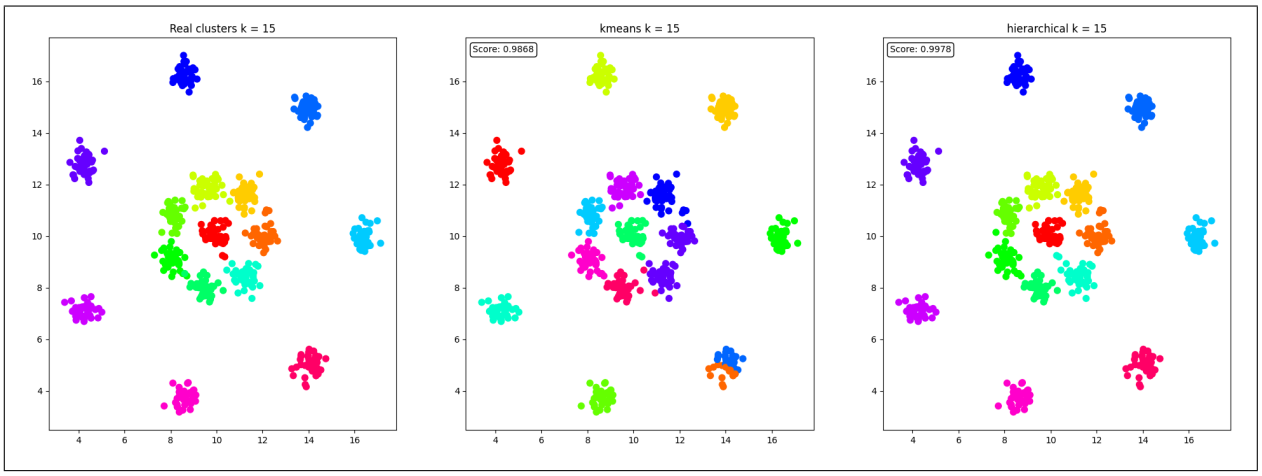
Rysunek 26: Dane 4, $k = 15$



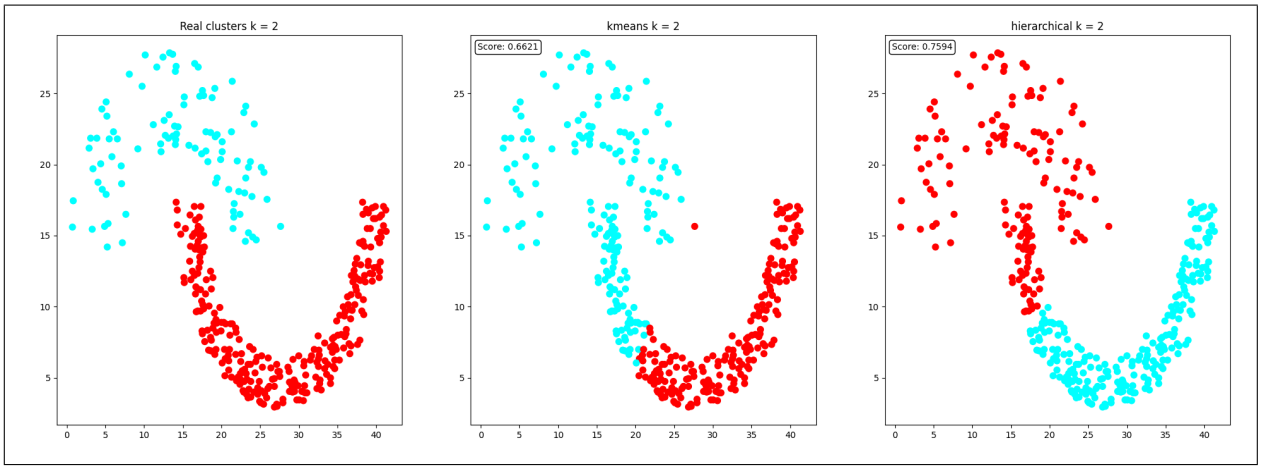
Rysunek 27: Dane 5, $k = 30$



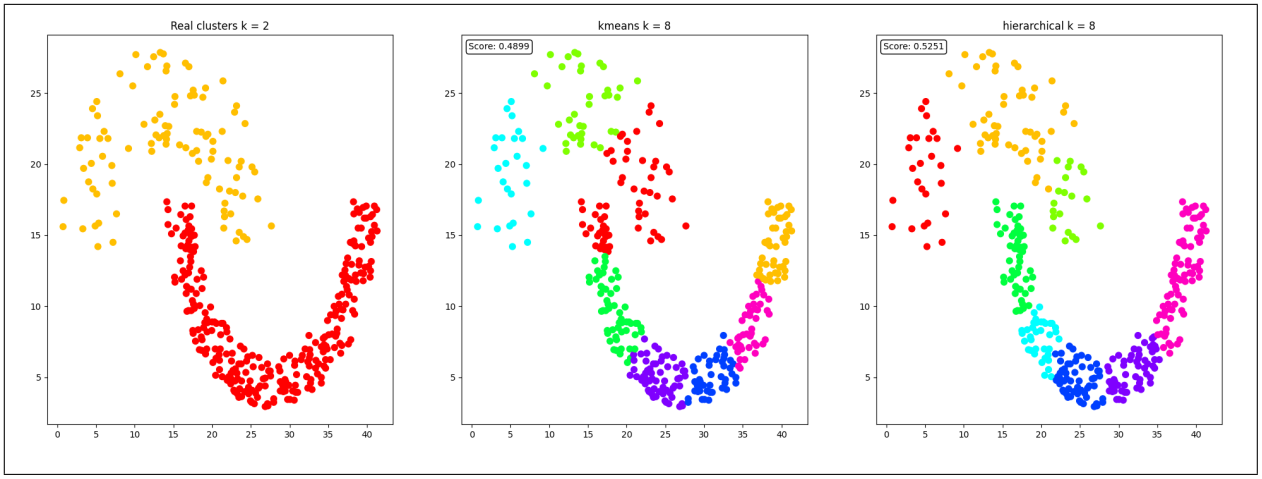
Rysunek 28: Dane 5, $k = 31$



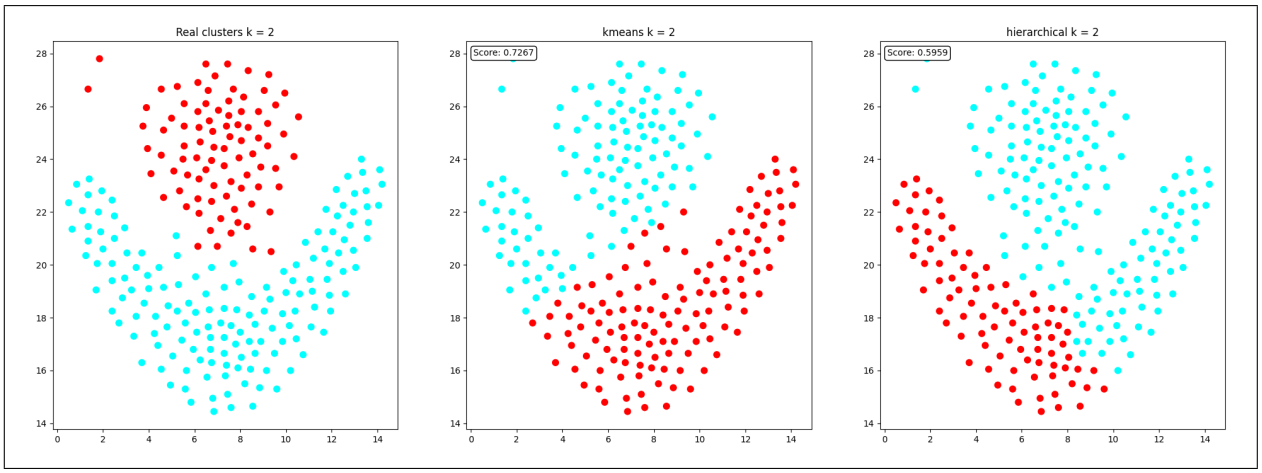
Rysunek 29: Dane 6, $k = 15$



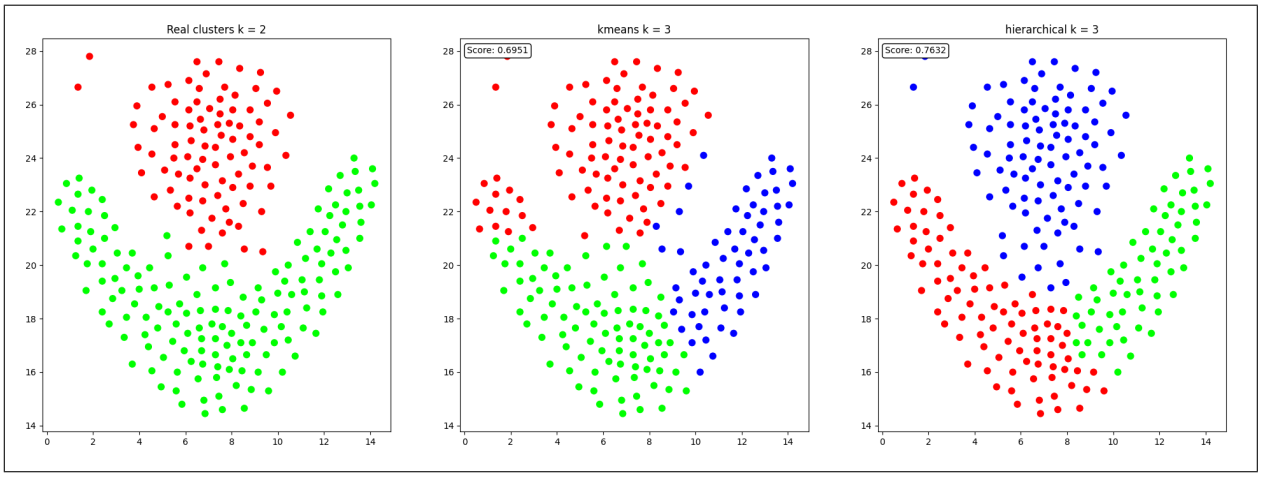
Rysunek 30: Dane 7, $k = 2$



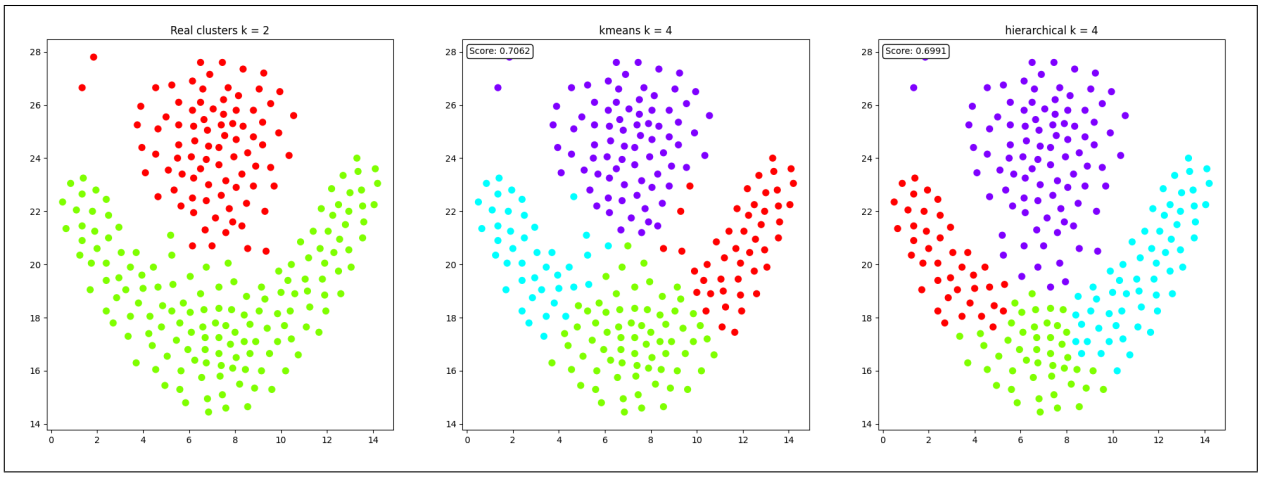
Rysunek 31: Dane 7, $k = 8$



Rysunek 32: Dane 8, $k = 2$



Rysunek 33: Dane 8, $k = 3$



Rysunek 34: Dane 8, $k = 4$

3.2 Pkiki dane_2D_n.txt - grupowanie spektralne

Dla zbiorów danych, których nie udało nam się sklasyfikować, zastosujemy klasteryzację spektralną. Na początku wypróbujemy metodę konstruowania macierzy sąsiedztwa opartą o metodę k najbliższych sąsiadów. Jak wspominałem w omówieniu, dobrym wyborem na k często jest liczba klastrów w zbiorze. Ponieważ poprzednie analizy doprowadziły do tego, że właściwie dla wszystkich zbiorów umieliśmy wskazać prawidłową liczbę klastrów, założymy, że już ją znamy.

Wykorzystując taką macierz, w opisany wcześniej sposób, przekształcimy przestrzeń punktów do innej przestrzeni wielowymiarowej, w której łatwiej będzie nam wyłapać skomplikowane zależności pomiędzy obiektami tego samego klastra.

O ile liczba wymiarów zazwyczaj odpowiada liczbie klastrów szukanych w danym zbiorze, o tyle wybór metody konstruowania macierzy sąsiedztwa oraz jej parametrów jest kwestią mocno indywidualną. Bazowo korzystałem z funkcji RBF, natomiast we wskazanych przypadkach wypróbowałem również inne metody. Heurystyki, które wskazałem w poprzednich rozdziałach, nie zawsze okazywały się najtrafniejszym wyborem, dlatego eksperymentowałem z bardzo różnymi ustawieniami. Poniżej przedstawiam różne wyniki wraz z ich analizą.

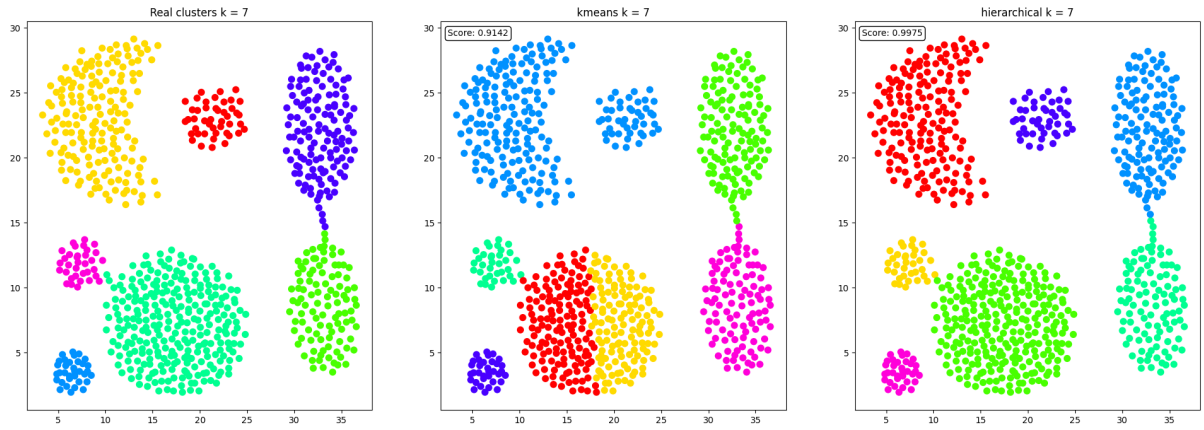
Dodatkowo, w tabelce zamieszczam dla każdego ze zbiorów, najlepszy regularnie uzyskiwany współczynnik *Score*, oraz algorytm, dla którego udało mi się go otrzymać:

Zbiór danych	**Wartość wskaźnika**	**Algorytm**
1	0.9975	Spektralny - KNN(20)
2	0.9465	Spektralny - RBF(1)
3	0.878	Spektralny - RBF(0.7)
4	1.0	Spektralny - RBF(1)
5	0.9951	Hierarchiczny
6	0.9978	Hierarchiczny
7	1.0	Spektralny - KNN(6)
8	0.9671	Spektralny - RBF(0.5)

Oczywiście, wskaźnik ten nie jest idealną metodą porównywania jakości algorytmów między zupełnie różnymi zbiorami danych, ale przynajmniej pozwala on odróżnić zbiory, które udało się praktycznie całkowicie poprawnie podzielić na klastry (wartość równa 1 lub bardzo bliska) od tych, których podział jest znacznie trudniejszym zadaniem, któremu badane algorytmy nie potrafiły sprostać (wartości niższe)

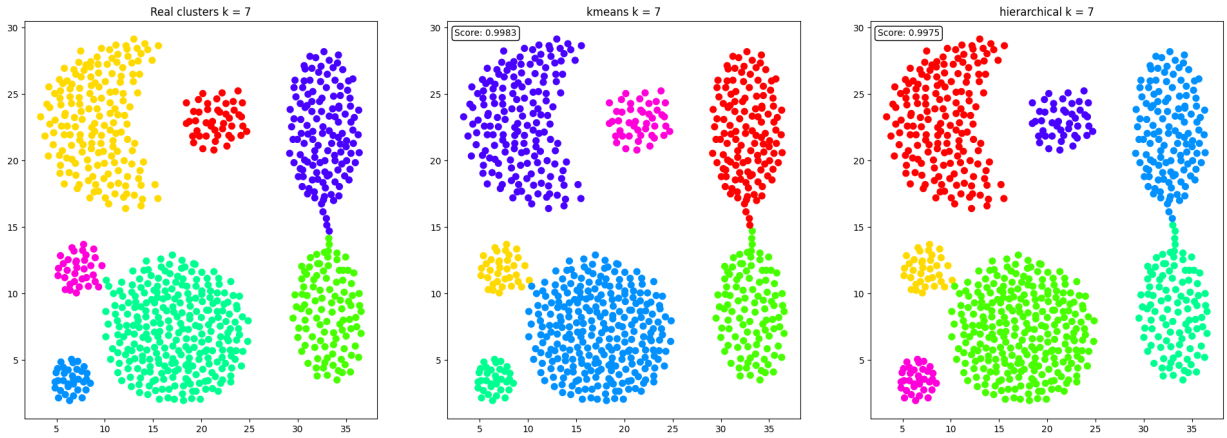
3.2.1 Dane_2D_1

Tutaj chciałem wypróbować macierz opartą o knn. Wybrałem współczynnik k o wartości zbliżającej się do rozmiaru małych klastrów, których poprzednie algorytmy nie potrafiły rozdzielić. Dało to naprawdę dobre wyniki, po transformacji i zastosowaniu klasyfikatora hierarchicznego, wyniki były niemal identyczne do rzeczywistych klastrów:



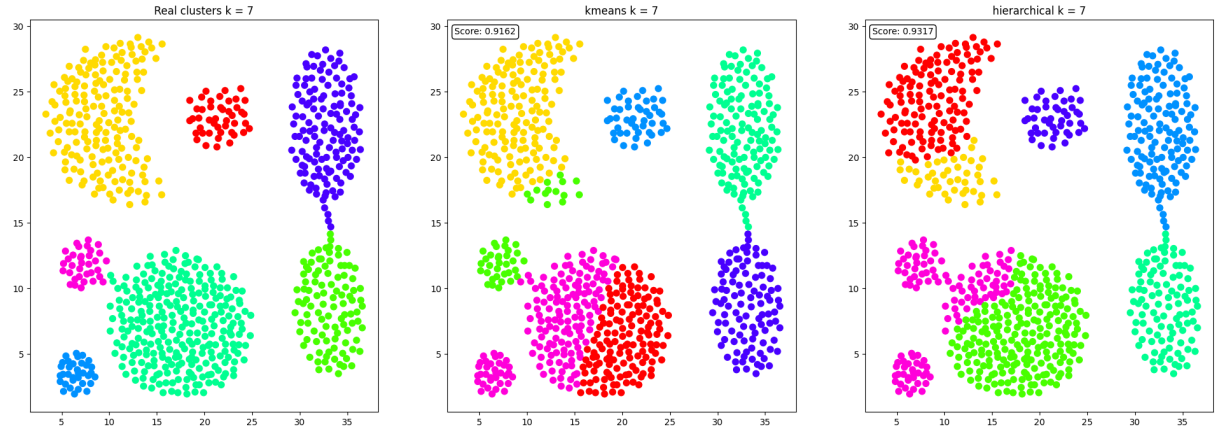
Rysunek 35: Dane 1 - KNN(20)

Oczywiście kmeans też radził sobie bardzo dobrze, jednak jego losowość mocno utrudnia przedstawienie tego, mimo korzystania z metody ++. Prawdopodobnie przy rzeczywistej potrzebie dokładnej klasteryzacji byłby on uruchamiany na wielu przebiegach i wybieralibyśmy ten najsensowniejszy. Poniżej jest przykład sytuacji, w której kmeans również świetnie sklasyfikował ten zbiór:



Rysunek 36: Dane 1 - KNN(20) - alternatywny przebieg

Wypróbowałem jeszcze metodę RBF, niestety nie zanotowałem podobnej skuteczności. Najlepsze były parametry oscylujące w okolicy $\gamma \approx 0.01$, jednak algorytm zazwyczaj prowadził do losowych, często nienaturalnych połączeń między klastrami, nie osiągając przy tym wyników nawet zbliżonych do poprzedniego algorytmu. Nie mniej jednak, udało się osiągnąć dużo lepsze wyniki niż proste klasyfikatory z poprzedniego działu.

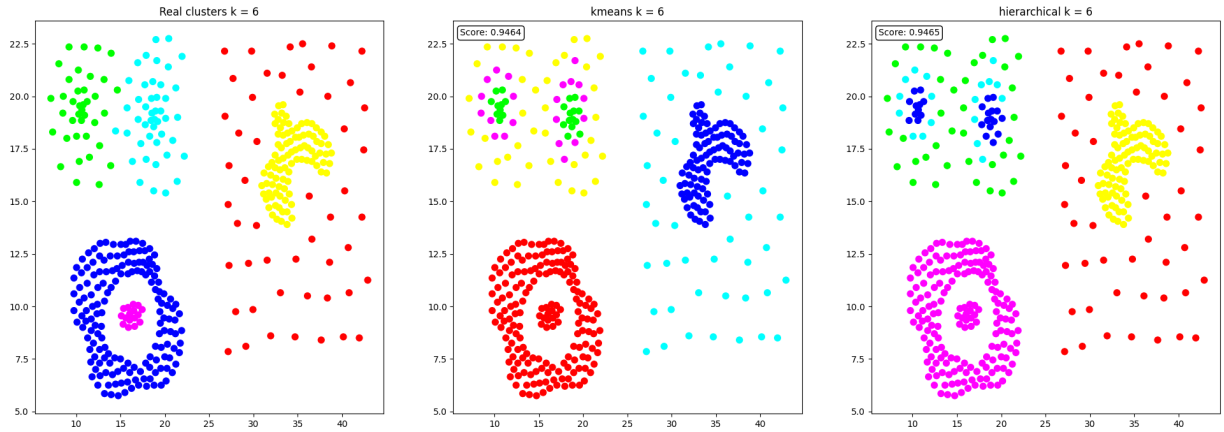


Rysunek 37: Dane 1 - RBF(0.01)

3.2.2 Dane_2D_2

Ten zbiór był dosyć trudny do klasyfikacji, ponieważ zawierał 2 ciężko odróżnialne klastry, które miały kompletnie różne charakterystyki i nie udało mi się znaleźć algorytmu, który byłby w stanie skutecznie przypisywać do klastrów obydwa zbiory.

Po pierwsze, chciałem odróżnić porozrzucane punkty z prawej storny od większej wysepki na środku. Do tego zadania najlepiej sprawdziła się metoda RBF z parametrem w okolicach 1.

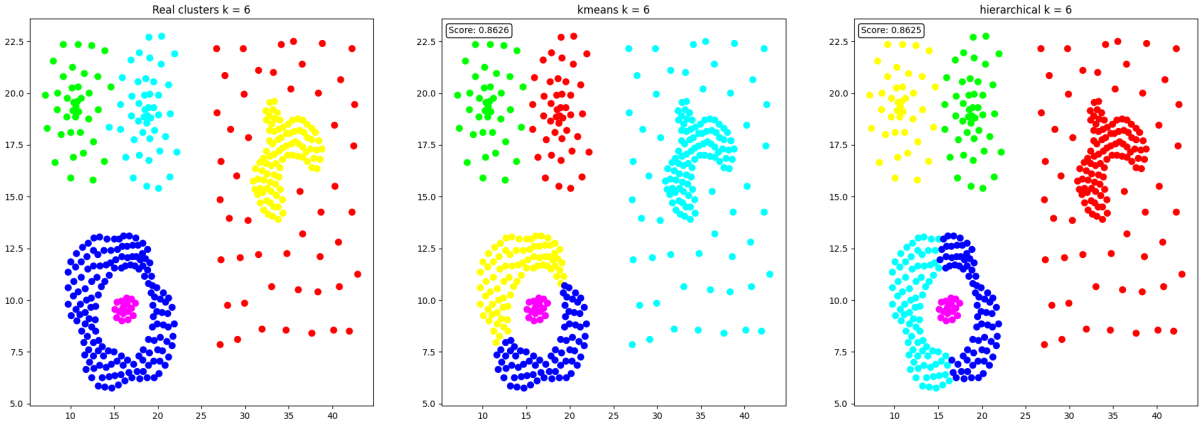


Rysunek 38: Dane 2 - RBF(1)

Widzimy jednak, że kompletnie nie radziła sobie z pozostałymi punktami, przypisywała mocno wątpliwe klastry.

Drugim problematycznym punktem był niewielki zbiór leżący wewnątrz innego klastru, metody oprate o klastry sferyczne w zasadzie nie miały szans go oddzielić od pozostałych. Tutaj ponownie zastosowałem strategię z poprzedniego zbioru, skorzystałem z macierzy knn przyjmując k aproksymujące liczbę punktów w problematycznym klastrze. Wyniki były zadowalające pod tym względem, że skutecznie oddzielały ten klaster, ale z kolei kompletnie nie radziły sobie one z porozrzucanymi punktami.

Podsumowując, w tym przypadku mieliśmy do czynienia z trudnym do jednoznacznej klasyfikacji zbiorem, ponieważ punkty miały zupełnie różny charakter. Znaleźliśmy jednak algorytmy, które radziły sobie nieźle z pojedynczymi sytuacjami.

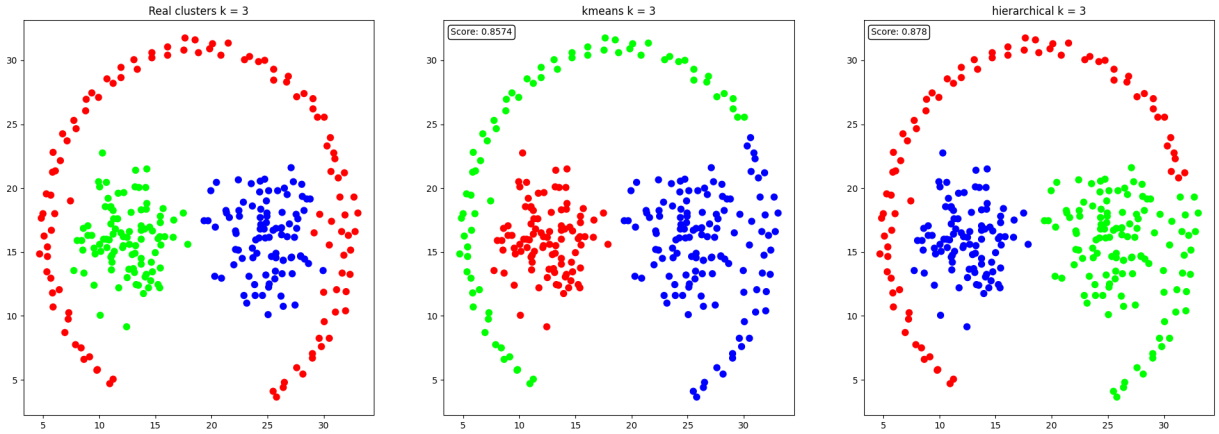


Rysunek 39: Dane 2 - KNN(10)

3.2.3 Dane_2D_3

W tym przypadku dość oczywistym i tak naprawdę jedynym problemem było oddzielenie łuku od kulistych klastrów. Zdecydowałem się wykorzystać funkcję RBF. Po dostosowywaniu parametru odkryłem, że najlepiej radzą sobie wartości $\gamma \in [0.5, 1]$, na rysunku przedstawione są wyniki dla $\gamma = 0.7$. Zbyt małe wartości powodowały mieszanie się kulistych klastrów do bocznych części łuku, zbyt duże prowadziły do klasyfikacji prawie wszystkich punktów w jeden wielki klaster.

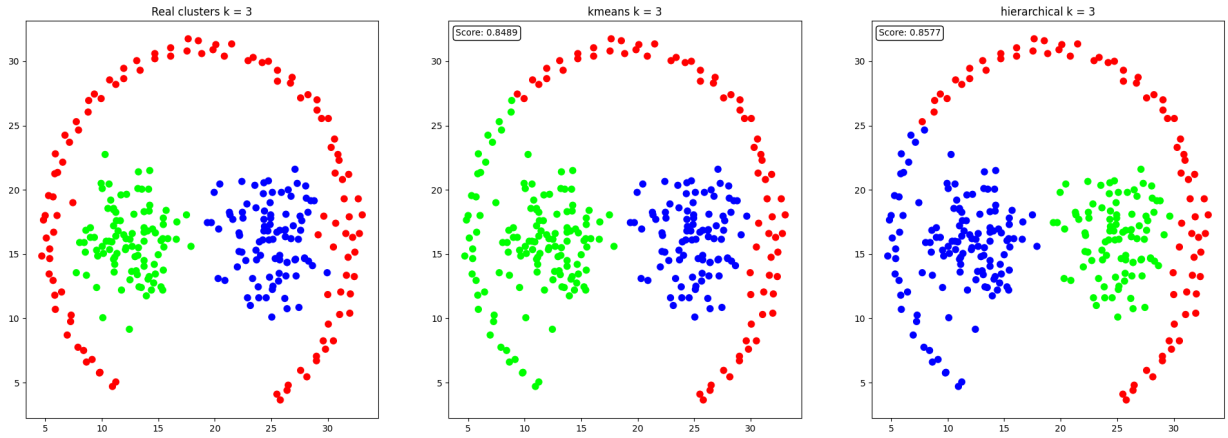
O ile taki parametr dość skutecznie oddzielał całą lewą stronę, o tyle w przypadku prawej, po której występuje znacznie więcej punktów pośredniczących, trudnych do sklasyfikowania, nie udało mi się w żaden sposób oddzielić tego fragmentu łuku od prawego kulistego klastra.



Rysunek 40: Dane 3 - RBF(0.7)

Bardzo ciekawa rzecz wydarzyła się, gdy wypróbowałem algorytm knn dla dość niskich wartości k (na rysunku $k = 6$). Bardzo ładnie wyizolował on prawy fragment łuku, jednak jak na złość nie poradził sobie ani trochę z lewym. Wyższe wartości parametru już niestety prowadziły do wmieszania się fragmentów łuku w sfery.

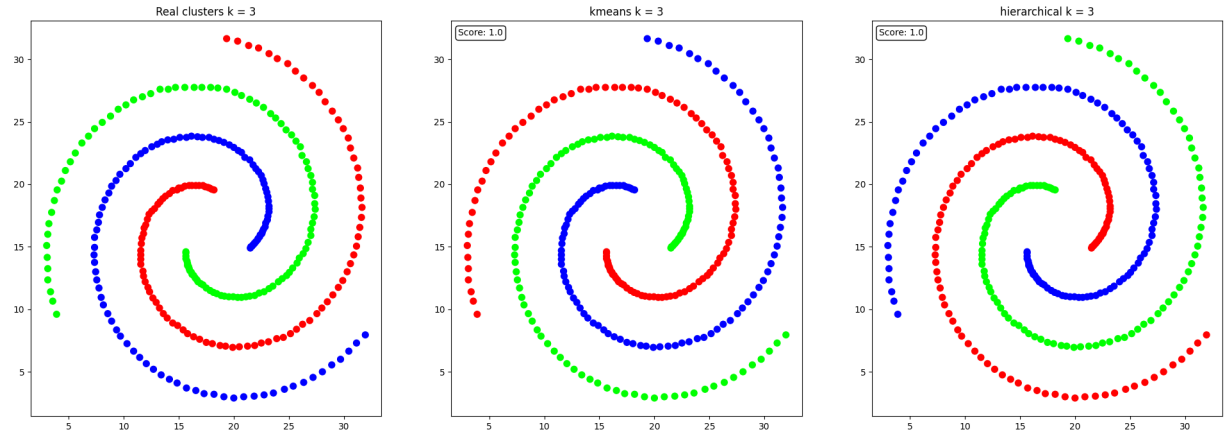
Teoretycznie moglibyśmy zastosować jakąś klasteryzację produktową, czyli do górnego fragmentu łuku dołączyć punkty połączone z nim w dowolnym z tych dwóch algorytmów, jednak wykorzystywalibyśmy już wtedy jakąś wiedzę, do której niekoniecznie mielibyśmy dostęp, gdyby danych nie dało się tak ładnie zwizualizować, np. gdyby zawierały więcej cech. Oddzielenie łuków jest trudnym zadaniem, ponieważ po bokach w zasadzie pasują one kształtem do znajdujących się w okolicy klastrów sferycznych, a jednak udało się w jakikolwiek sposób je rozdzielić.



Rysunek 41: Dane 3 - KNN(6)

3.2.4 Dane_2D_4

W tym przypadku w zasadzie od razu uzyskałem idealną klasteryzację stosując funkcję RBF z parametrem $\gamma = 1$:



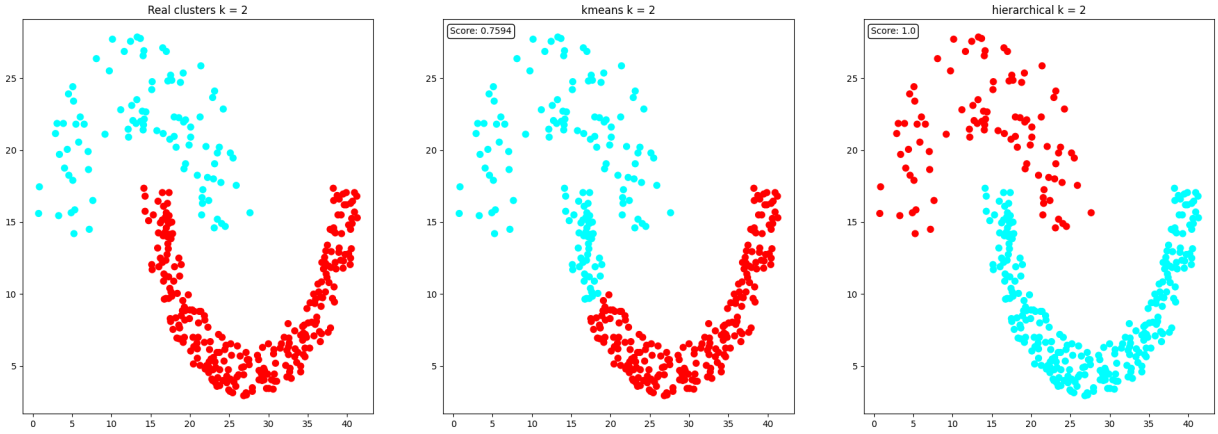
Rysunek 42: Dane 4 - RBF(1)

Oczywiście, kmeans nie zawsze zwracał wynik stuprocentowy, jednak udawało mu się to osiągnąć mniej więcej raz na 3-4 próby, wyniki zależał więc mocno od początkowego wyboru centroidów.

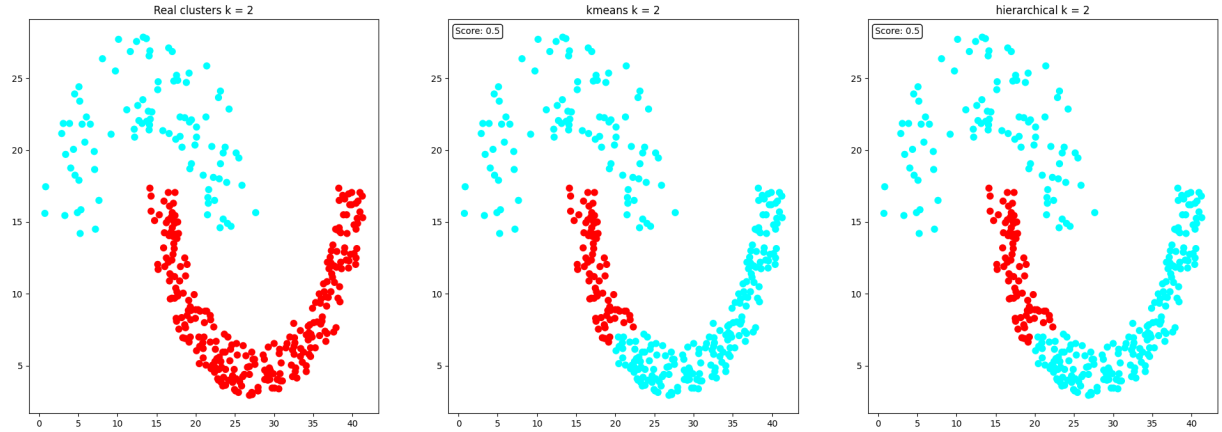
Stuprocentową skuteczność możemy wytłumaczyć faktem, że zbiory były naprawdę ładnie podzielonę, gołym okiem widać w nich granicę. Prostsze algorytmy nie były jednak w stanie jej wyłapać, dopiero transformacja do innej przestrzeni (w tym przypadku również dwuwymiarowej) pozwoliła na oddzielenie klastrów od siebie.

3.2.5 Dane_2D_7

W tym przypadku również zadziałała redukcja do innej przestrzeni dwuwymiarowej, tym razem skorzystałem jednak z metody knn, z parametrem $k = 10$, ale dosyć szerokie spektrum wartości k zwracało podobne wyniki. Najbardziej zaskoczyło mnie, jak niewielka zmiana parametru k w tej metodzie wpływała na wyniki. Dla $k = 6$, wyniki klasteryzacji hierarchicznej był perfekcyjny, jednak dla $k = 5$ algorytm kompletnie sobie nie poradził. Pokazuje to, dlaczego warto przetestować szeroką gamę wartości. Kmeans również był w stanie osiągnąć skuteczność stuprocentową, ale udawało mu się to znacznie rzadziej niż w poprzednim przypadku.



Rysunek 43: Dane 7 - KNN(6)

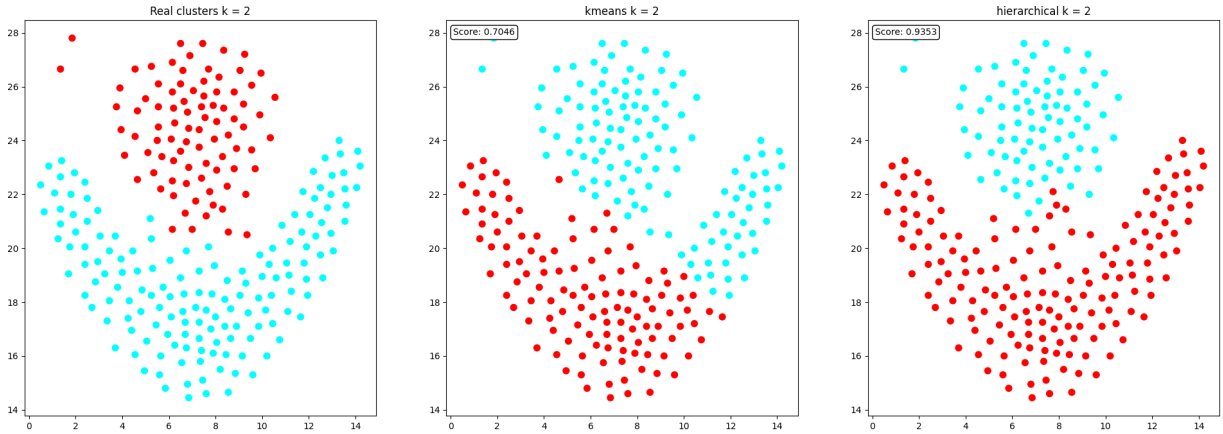


Rysunek 44: Dane 7 - KNN(5)

Pozostałe algorytmy nie poradziły sobie najlepiej. Najczęściej występującym problemem było zakwalifikowanie gęstego lewego fragmentu dolnego łuku do klastru górnego, czego przykład widzimy w przestawionym wyżej wyniku algorytmu kmeans dla $k = 6$. Zbiory były dosyć proste do odróżnienia, jednak w jednym z nich znajdowało się o wiele mniej danych, co prawdopodobnie doprowadzało do błędów czynionych przez klasyfikatory.

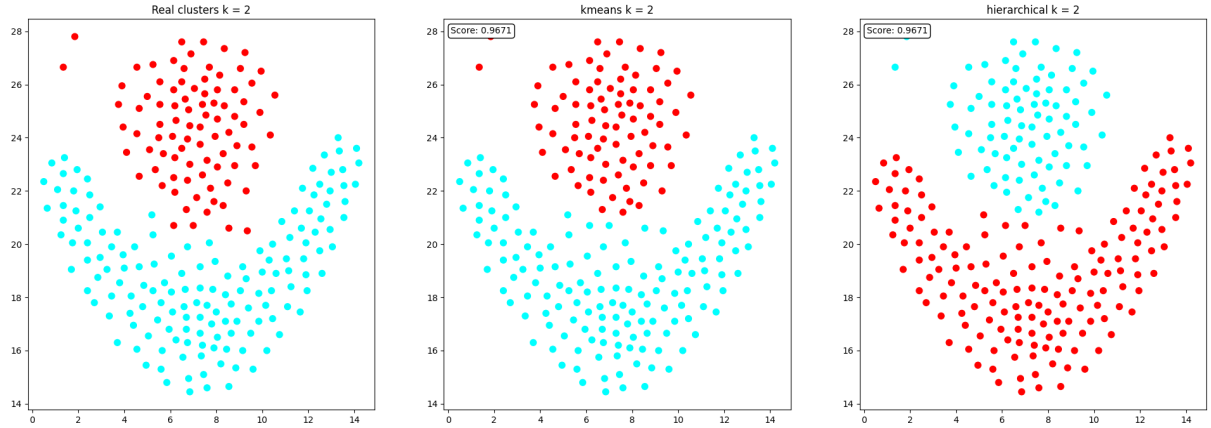
3.2.6 Dane_2D_8

W tym przypadku pierwszy raz o wiele dokładniej zbadałem metodę konstrukcji opartą o dystans krótszy niż zadany parametr ϵ. Udało mi się odnaleźć lokalne maksimum dla $\epsilon = 24$. Ponownie, niesamowite było to jak bardzo konstrukcja macierzy sąsiedztwa jest czuła na niewielkie zmiany. Dla odchyleń parametru $\epsilon \approx 23.9$ oraz $\epsilon \approx 24.1$ klasyfikator osiągał już wynik *Score* na poziomie 0.6, co stanowi znaczne pogorszenie jego jakości. Prawdopodobnie był to dystans pomiędzy punktami stanowiącymi boczne brzegi dolnego półksiężyca do ich najbliższych punktów z klastra sferycznego, ich połączenie w macierzy doprowadziło do dużych błędów klasyfikacji.



Rysunek 45: Dane 8 - E-DIST(24)

Jeszcze lepsze wyniki uzyskałem przy pomocy metody RBF z parametrem $\gamma = 0.5$. Tutaj już klasyfikator był nieco mniej czuły, okolice tego parametru również dawały podobne wyniki. W tym przypadku punkty również



Rysunek 46: Dane 8 - RBF(0.5)

były mapowane do innej przestrzeni dwuwymiarowej, można więc ładnie przedstawić przebieg mapowania oraz iteracje algorytmów na zmapowanej przestrzeni.

3.3 Plik rp.data

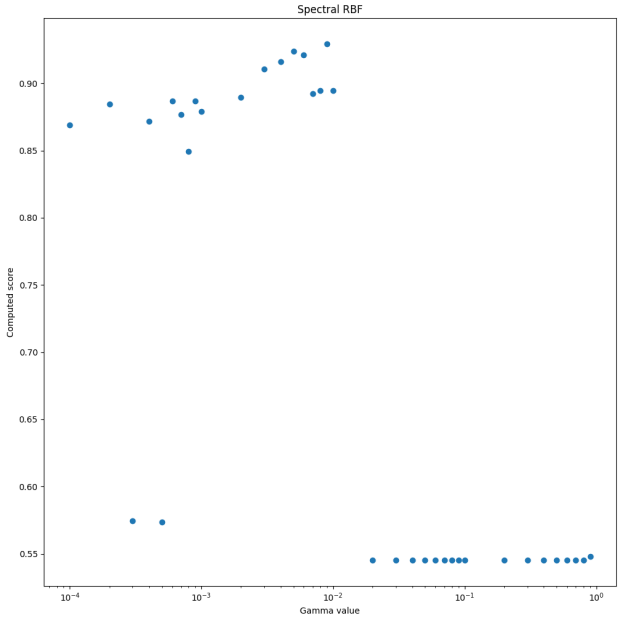
W przypadku tych danych wiemy z góry, że są podzielone na 2 klasy. Możemy więc pominąć proces szukania najlepszej liczby klastrów za pomocą chociażby metody sylwetkowej. Problemem może być wizualizacja danych, dlatego po prostu uruchomię poprzednio przedstawione algorytmy i przedstawię uzyskane wskaźniki *Score*.

- Średnią wartością wskaźnika uzyskanego przy pomocy metody kmeans++ na kilkudziesięciu niezależnych przebiegach jest 0.924
- Dla algorytmu grupowania hierarchicznego, wartość wskaźnika po określeniu liczby klastrów jako 2 wyniosła 0.938

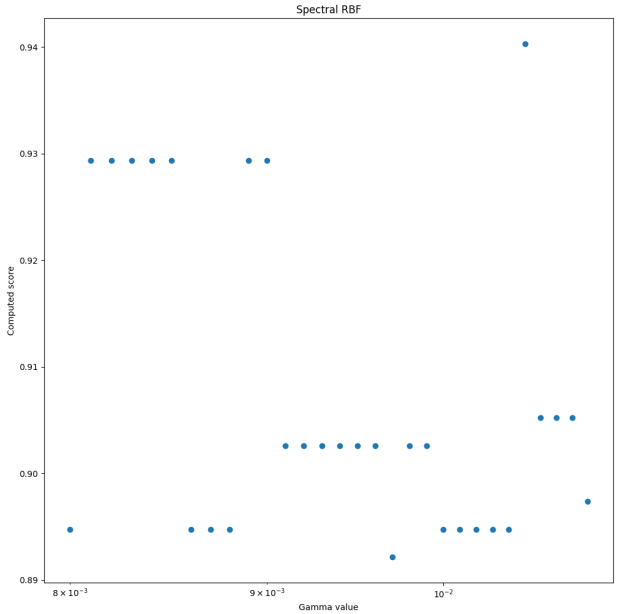
Następnie, korzystamy z metody spektralnej. Dla każdej metody konstrukcji macierzy szukamy najlepszej wartości parametru, mając na uwagę, że dla mniejszych zbiorów danych algorytmy są często wrażliwe na nawet niewielkie zmiany. Najlepsze wyniki dla poszczególnych metod wyglądają następująco:

- Metoda RBF dla parametru $\gamma = 0.0105$ uzyskała wynik 94.03. Jest to jednak bardzo specyficzna wartość parametru, która nie wiedzieć czemu wybiła się znacząco w swoim otoczeniu. W okolicach optimum wynik oscylował wokół 92.93
- Metoda KNN generalnie uzyskiwała dobre, zbliżone do siebie wyniki dla dostatecznie wysokich wartości k, najlepszym wynikiem, który kilkukrotnie wystąpił, było 92.12
- Metoda E_DISTANCE była bardzo wraźliwa na niewielkie zmiany, w najlepszym punkcie uzyskaliśmy wartość 91.6, jednak bez dogłębnej analizy danych ciężko byłoby ją znaleźć

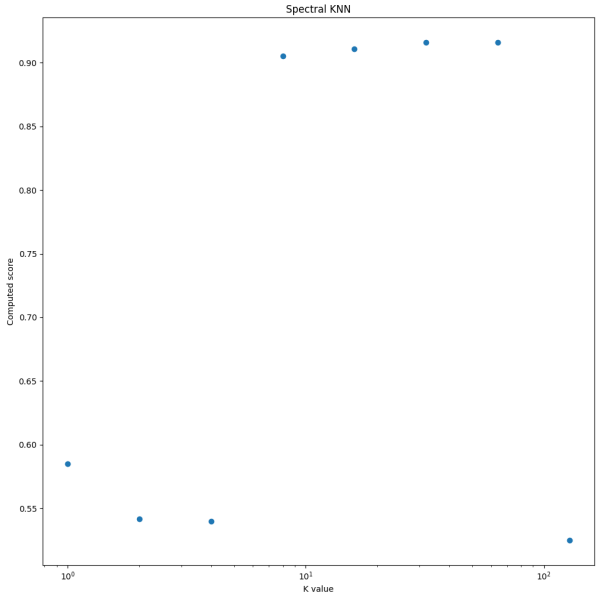
Generalnie najlepiej poradziło sobie grupowanie hierarchiczne. Metody spektralne są dość problematyczne w implementacji, kiedy nie możemy ładnie zwizualizować ich wyników. Jak zauważyliśmy, niewielkie zmiany w parametrach metody doprowadzały do całkowicie różnych wyników



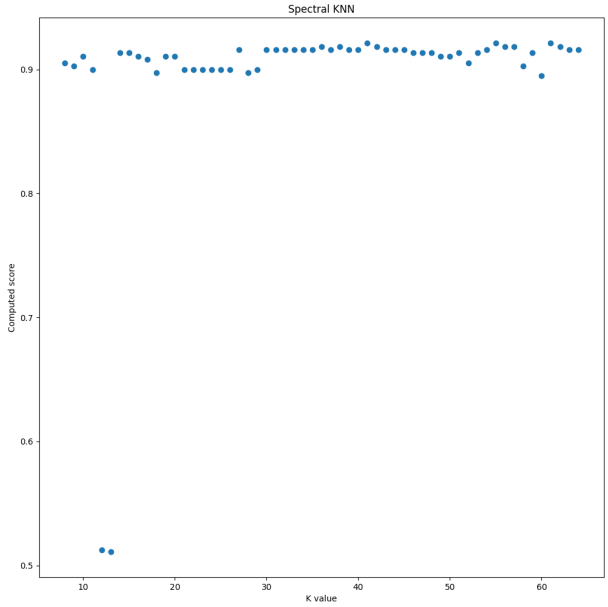
Rysunek 47: RBF - wyszukiwanie szeroko



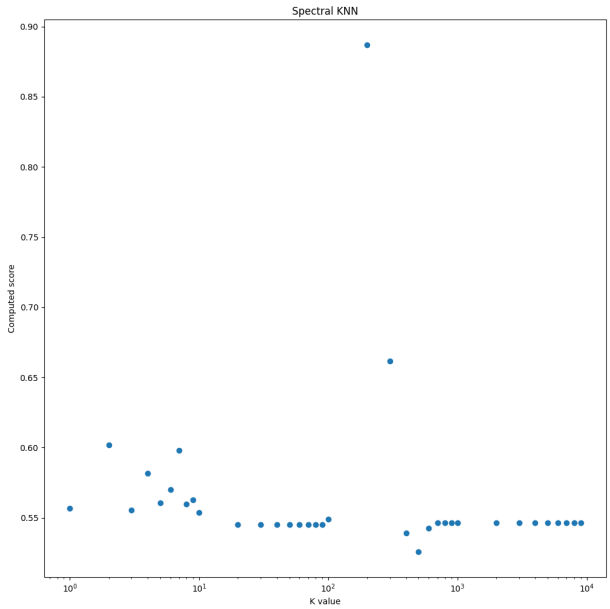
Rysunek 48: RBF - wyszukiwanie w okolicach ekstremum



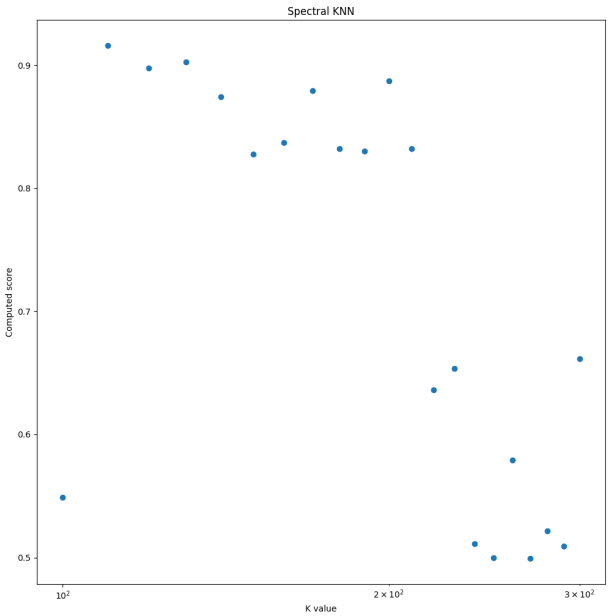
Rysunek 49: KNN - wyszukiwanie szeroko



Rysunek 50: KNN - wyszukiwanie w okolicach ekstremum



Rysunek 51: EDIST - wyszukiwanie szeroko

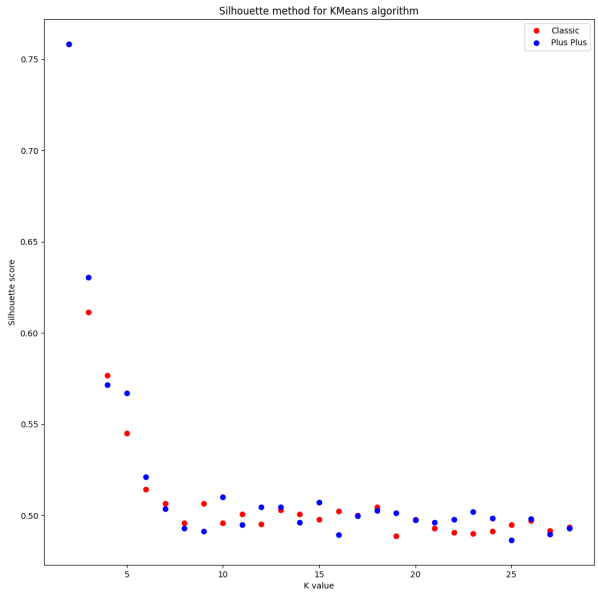


Rysunek 52: EDIST - wyszukiwanie w okolicach ekstremum

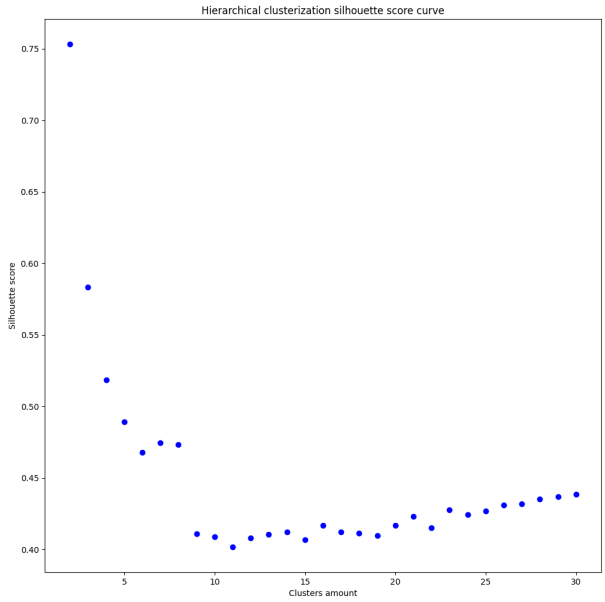
3.4 Plik Dane_18D.txt

Po pierwsze, w przeciwieństwie do danych z pozostałych plików, zastosujemy skalowanie do przedziału [0,1], ponieważ zbiory wartości poszczególnych cech znacząco się od siebie różnią.

W tym zbiorze dużym wyzwaniem jest brak rzeczywistego podziału na klastry, którym moglibyśmy się posłużyć w celu badania jakości algorytmu. Spróbujemy więc po prostu wyznaczyć ich optymalną liczbę stosując metodę sylwetkową.

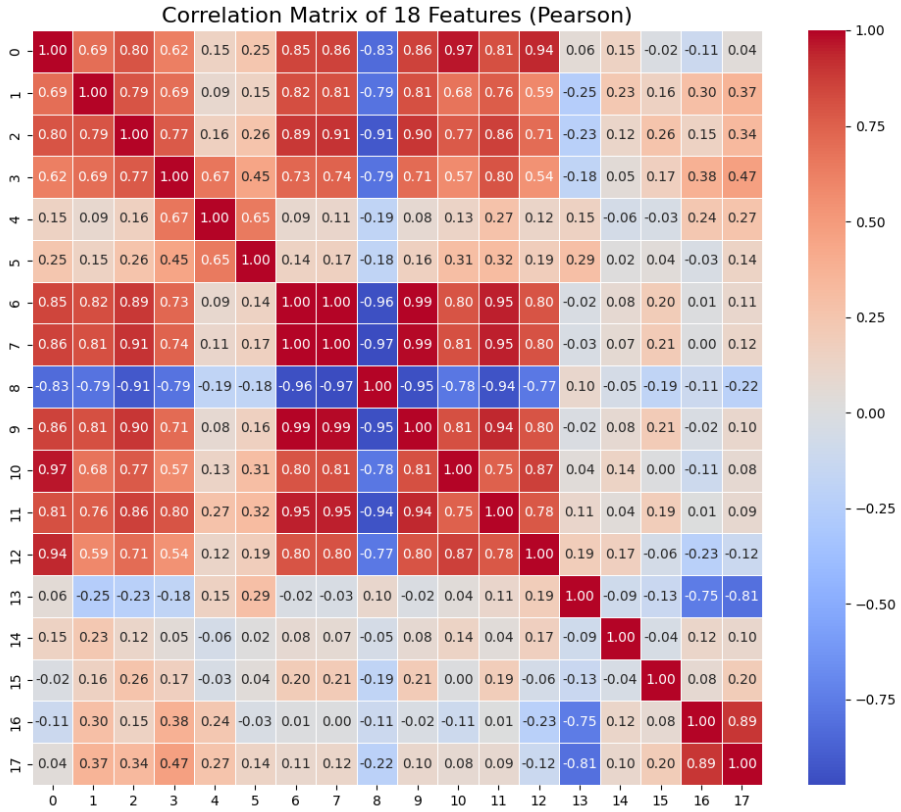


Rysunek 53: 18D kmeans - analiza współczynnika



Rysunek 54: 18D grupowanie hierarchiczne - analiza współczynnika

Obydwa wyniki mocno sugerują, że dane z pliku podzielone są na 2 klastry. W takim wypadku pomyślałem, że być może sporo spośród ich 18 cech jest redundantnych, i może udałoby się zredukować te punkty do mniejszej liczby wymiarów, aby dało się je ładnej przedstawić. Poniżej przedstawiona jest macierz korelacji cech:



Rysunek 55: 18D grupowanie hierarchiczne - analiza współczynnika

Jak widać, duża liczba cech jest ze sobą silnie powiązanych. W szczególności, cała grupa $[0, 3] \cup [6, 12]$ jest mocno skorelowana liniowo ze sobą, podobnie jak $\{13, 16, 17\}$.

Przyglądając się głębiej tabelce wydaje się, że 18 cech można podzielić na 6 silnie zależnych od siebie grup:

$$\{0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12\} \cup \{13, 16, 17\} \cup \{4\} \cup \{5\} \cup \{14\} \cup \{15\}$$

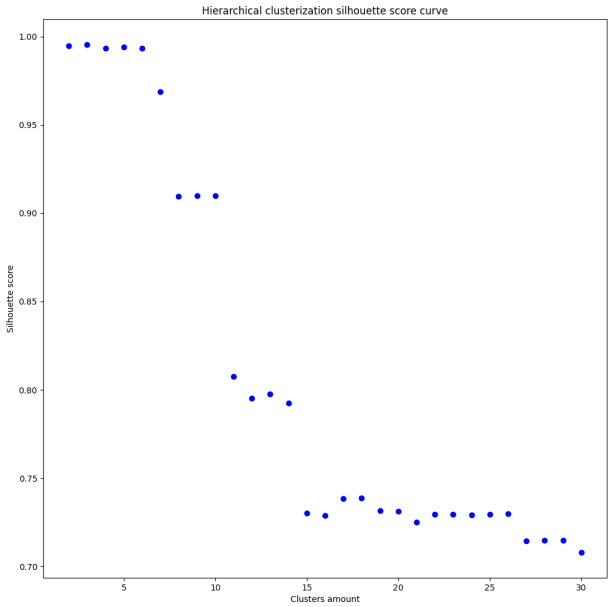
Postanowiłem więc skorzystać z grupowania spektralnego, żeby przetransformować przestrzeń do nowej, sześciowymiarowej przestrzeni. Aby obliczyć parametr γ skorzystałem z heurystyki:

$$\gamma = \frac{1}{d_k^2}$$

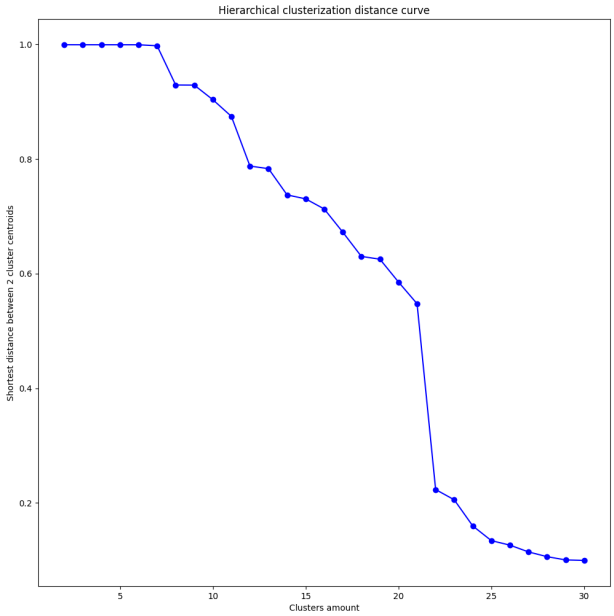
gdzie d_k to średnia po średnich odległościach punktu od pozostałych punktów w zbiorze. Na przeskalowanych danych, wyszło mi $\gamma \approx 10$. Udało się uzyskać o wiele lepsze wyniki współczynnika sylwetkowego, niż przed transformacją. Na kolejnej stronie zamieściłem wykres po skorzystaniu z grupowania hierarchicznego na powstałym nowym zbiorze danych. Jak widać, dla podejrzewanych wcześniej niskich wartości k wartość współczynnika jest bliska 1, co oznacza, że dane po transformacji można dość łatwo liniowo odseparować od siebie, tworząc przy tym pewną liczbę oddzielnych klastrów.

Tutaj pierwszy raz heurystyka wskazała naprawdę dobrą wartość parametru. W poprzednich przypadkach, do których próbowałem ją zastosować, zazwyczaj obstawiana była zbyt niska wartość, i musiałem szukać optymalnej na inne sposoby, np. porównując wyniki na wykresie.

Analizując wyniki z wykresu dystansu wnioskujemy, że dane bardzo ładnie podzieliły się na 7 rozłącznych klastrów, które po transformacji znajdowały się daleko od siebie, potem widzimy już spory spadek wartości funkcji. Podejrzewamy więc, że w rzeczywistości tworzą one liczbę klastrów w przedziale $k \in [2, 6]$



Rysunek 56: 18D grupowanie hierarchiczne - analiza współczynnika po transformacji przestrzeni



Rysunek 57: 18D grupowanie hierarchiczne - analiza dystansu 2 najbliższych klastrów po transformacji przestrzeni

4 Animacje

Do katalogu Miniprojekt4/videos dołączę wizualizację przebiegów algorytmów, o których wspominałem w różnych sekcjach raportu. Znajdują się tam:

- Przykładowe przebiegi algorytmu k-means++ na zbiorach dwuwymiarowych dla losowego wyboru centroidów,
- Przebiegi algorytmu grupowania hierarchicznego dla zbiorów dwuwymiarowych od początku, gdzie każdy punkt leżał w osobnym klastrze, do momentu, aż wszystkie zostały połączone w jeden,
- Dla zbiorów, w przypadku których dobre wyniki dała transformacja spektralna do innej przestrzeni dwuwymiarowej, załączam również porównanie punktów przez i po transformacji oraz przebiegi dwóch poprzednio wymienionych algorytmów na nowej przestrzeni. Tutaj w niektórych przypadkach algorytmy k-means produkowały bardzo różniące się rezultaty, szczególnie, gdy liczba klastrów była bardzo mała (np. 2). Starałem się więc wybrać ładne przebiegi. Nie są to niewiadomo jak rzadkie sytuacje, zdarzały się raz na kilka prób losowania.