Grupowanie szeregów czasowych

(Time Series Clustering)

Andrzej Kołacz

Praca inżynierska

Promotor: dr hab. Piotr Wnuk-Lipiński

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

3 września 2018

Streszczenie

Celem niniejszej pracy dyplomowej jest implementacja, zastosowanie oraz zbadanie narzędzia służącego do grupowania podciągów szeregu czasowego, bazując na artykule *"Selective Subsequence Time Series clustering"* autorstwa S. Rodpongpun, V. Niennattrakul oraz C. A. Ratanamahatana. Został napisany program, który zgodnie z wyżej wymienioną pracą naukową, znajduje grupy podobnych podciągów jednowymiarowego szeregu czasowego, a także został on wzbogacony o możliwość grupowania danych wielowymiarowych.

W algorytmie użyto metody *sliding window*, która pozwala na wyizolowanie z wejściowego szeregu czasowego wszystkich podciągów o długości z zadanego przedziału. Następnie kierując się pewnym kryterium zachłannym, algorytm wybiera jedną z trzech funkcji konstruujących wynikowe grupowanie. Cały proces wspiera idea kodowania wejściowego szeregu czasowego.

The purpose of present Engineer's thesis was to implement, apply and investigate a tool for subsequence time series clustering based on algorithm, which was featured in the article *"Selective Subsequence Time Series clustering"* written by S. Rodpongpun, V. Niennattrakul and C. A. Ratanamahatana. There was created a program that, according to the thesis mentioned above, finds groups of similar subsequence time series within single univariate time series. The program was also enriched with the ability to cluster multivariate time series.

The *sliding window* method was used to extract all subsequences of the length from the given interval from the input time series. Then algorithm is lead by some greedy criterion and chooses one of three functions that builds the consecutive clusters. The whole process is supported by an idea of data encoding.

Spis treści

Rozdział 1.

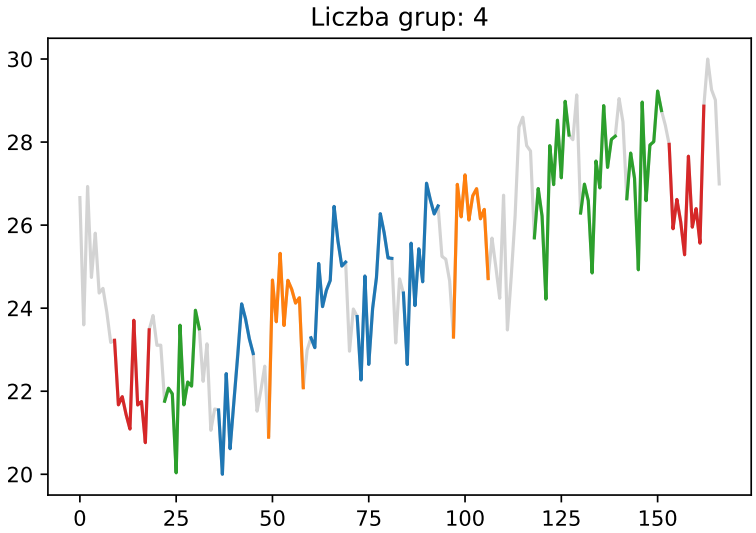
Wprowadzenie

Grupowanie, inaczej *analiza skupień*, jest jednym z podstawowych zadań eksploracji danych. Polega na wyznaczaniu pewnych podobieństw wśród elementów rozpatrywanej bazy danych, poprzez rozdzielanie ich na zbiory nazywane grupami. Od tak skonstruowanego rozwiązania wymagamy, by dane należące do jednej grupy były jak najbardziej zbliżone do siebie, podczas gdy między zbiorami jesteśmy w stanie dostrzec istotne różnice. Celem takiej metody może być np. wstępna analiza, w której wyodrębniamy skupiska jednorodne względem pewnej własności. Dzięki temu możemy skuteczniej analizować je za pomocą kolejnych narzędzi statystycznych oraz wskazywać na ich zasadnicze cechy. Ponadto przy użyciu grupowania łatwiejsze staje się odkrywanie struktury samych danych. Właśnie z tego powodu to zagadnienie było wielokrotnie rozpatrywane w dziedzinie *szeregów czasowych* [1, 3]. Grupowanie może dostarczać nam informacji o pewnych zawartych w nich wzorcach. W związku z intensywnym rozwojem technologii i wszechobecnością danych zależnych od czasu odnajduje to swoje zastosowanie w wielu obszarach. Szeregami czasowymi jesteśmy w stanie reprezentować zjawiska zarejestrowane na filmach, nagraniach audio czy ilustracjach [1], giełdowe kursy akcji, procesy biologiczne, warunki atmosferyczne, dzieła tekstowe i wiele innych. Istotne zatem jest opracowanie metod, które pomogą nam wydobywać z tych danych jak najistotniejsze prawidłowości. Spośród podejść do grupowania szeregów czasowych wyróżniamy dwa główne:

- grupowanie odrębnych szeregów czasowych — za bazę danych przyjmuje się zbiór złożony z *indywidualnych* szeregów czasowych, dotyczących często jednego zagadnienia, np. grupowanie danych reprezentujących wyniki sprzedaży w ciągu roku dla każdego z produktów, dzięki któremu możemy wskazywać na zależności pomiędzy zapotrzebowaniem i charakteryzować pewne okresy,

- grupowanie podciągów w obrębie jednego szeregu czasowego — bazą danych staje się zbiór różnych podciągów wydobytych z tego samego szeregu czasowego, np. grupowanie przedziałów danych z elektrokardiogramu (EKG), dzięki któremu możemy wykrywać anomalie oraz drobne zaburzenia [1].

O ile pierwsze zagadnienie cieszyło się większą popularnością i opracowane zostały efektywne metody rozwiązujące ten problem, to drugie do niedawna nie dysponowało poprawnym algorytmem. Dopiero w roku 2012 ukazała się praca *"Selective Subsequence Time Series clustering"* autorstwa S. Rodpongpun, V. Niennattrakul oraz C. A. Ratanamahatana, w której opisana została metoda produkująca poprawne rezultaty [1, 3]. W mojej pracy przedstawię sposób jej działania oraz zweryfikuję skuteczność swojej implementacji. Dodatkowo rozszerzę algorytm o możliwość działania na wielowymiarowych szeregach czasowych, które stanowią istotną część współcześnie rejestrowanych danych.



Rysunek 1.1: Przykładowy rezultat działania algorytmu na danych jednowymiarowych [5].

Dalsza część pracy jest zorganizowana w następujący sposób. Rozdział 2. zawiera formalny opis problemu oraz definicje podstawowych pojęć użytych w późniejszych rozważaniach. W rozdziale 3. przeprowadzona jest szczegółowa analiza algorytmu. Rozdział 4. uwzględnia potrzebne modyfikacje, dzięki którym możliwe jest grupowanie wielowymiarowych szeregów czasowych. W rozdziale 5. znajdują się wyniki doświadczeń podkreślające zalety oraz wady metody.

Implementacja, której najistotniejsze fragmenty wzbogacają rozważania teoretyczne w tej pracy, została zrealizowana w języku *Python 3.6* (dystrybucja Anaconda), a wszelkie eksperymenty zostały przeprowadzone w środowisku *Jupyter Notebook*.

Rozdział 2.

Teoretyczny opis zagadnienia

Formalnie szeregiem czasowym nazywamy realizację *procesu stochastycznego*, którego dziedziną jest czas, jednak w niniejszych rozważaniach utożsamiamy go z jego reprezentacją, tj. z ciągiem informacji uporządkowanych w czasie.

Definicja 1. *Szeregiem czasowym (jednowymiarowym)* T o długości n nazywamy ciąg liczb $t_i \in \mathbb{R}$, taki że $T = (t_1, t_2, ..., t_n)$. [1]

Definicja 2. *Podciągiem* długości m szeregu czasowego T o długości n nazywamy szereg czasowy $T_{i,m} = (t_i, t_{i+1}, ..., t_{i+m-1})$, gdzie $1 \leqslant i \leqslant n - m + 1$, $m < n$.

Problem (Grupowanie)**.** Mając dany szereg czasowy, chcemy znaleźć przeciwdziedzinę suriekcji z pewnego zbioru parami rozłącznych podciągów szeregu czasowego w zbiór *grup*. Dowolną grupę (*klaster*) stanowić będzie niepusty zbiór podciągów, które uważamy za podobne względem siebie. Od wynikowej *klasteryzacji* wymagamy, żeby spełniała określony warunek.

Rozpatrując przestrzenie afiniczne nad ciałem liczb rzeczywistych, naturalne jest przyjęcie, że podobieństwo dwóch elementów jest ujemnie skorelowane z dzielącą je odległością. W dziedzinie szeregów czasowych autorzy często skłaniają się ku wyborowi metryki euklidesowej:

Definicja 3. *Odległością* między dwoma szeregami czasowymi T_i oraz T_j nazywamy wartość $dist(T_i, T_j) = \sqrt{\sum_{k=1}^{n}(t_{i,k} - t_{j,k})^2}$.

Na podstawie wyników wielu doświadczeń jest ona uważana za konkurencyjną bądź skuteczniejszą od bardziej złożonych miar [2]. Co więcej, za wybraniem jej przemawia także szybkość wykonywania obliczeń, która ma znaczenie przy dużych zbiorach danych.

Spośród wszystkich możliwych grupowań podciągów rozwiązaniem nazywamy to, które maksymalizuje pewną *miarę jakości grupowania*. Jej wybór jest kluczowy dla sposobu działania algorytmu, a co za tym idzie, decyduje o uzyskanych wynikach.

Przykładem takiej miary może być stosunek sumarycznego podobieństwa elementów w obrębie grup do sumarycznego podobieństwa wszystkich par elementów pochodzących z różnych grup. Współgra to z intuicyjnym stwierdzeniem, że im bardziej zbliżone do siebie są elementy w grupach a różne między grupami, tym dany rezultat jest lepszy. Jednak taki wybór jest niedopuszczalny ze względu na złożoność obliczeniową, niezależnie czy mówimy o podciągach szeregów czasowych, czy o mniej złożonych danych. Istnieją oczywiście miary, które sprawdzają się w zastosowaniach praktycznych. Wiele z nich, tak jak i opisywany przeze mnie algorytm, opiera się elementach reprezentujących daną grupę, tzw. *wektorach kodowych*, które w wielu opisach są utożsamiane z *centroidami*.

W metodach klasteryzacji centroidy (*centra*) są często średnią ze wszystkich członków grupy (niekoniecznie wtedy muszą należeć do klastra) lub rzadziej, faktycznym członkiem najbliższym tej średniej. Zazwyczaj stanowią one część rozwiązania jako dodatkowe wartości cechujące i ściśle powiązane z wynikowymi zbiorami. Na potrzeby dalszych rozważań podamy definicję *średniej ważonej*, którą będziemy wykorzystywać przy wyliczaniu centroidów.

Definicja 4. *Średnią ważoną* z dwóch podciągów $P = (p_1, p_2, ..., p_m)$ i $Q = (q_1, q_2, ..., q_m)$ nazywamy ciąg $R = (r_1, r_2, ..., r_m)$, taki że $r_i = \frac{p_i\omega_p + q_i\omega_q}{\omega_p + \omega_q}$, gdzie ω_p i ω_q to *wagi* odpowiednio podciągów P i Q.

Podejść do uzyskania znaczących wyników w dziedzinie szeregów czasowych było kilka. Niestety pierwsze metody ze względu na swoją gorliwość, tj. grupowanie wszelkich możliwych podciągów, skutkowały uzyskiwaniem centroidów o sinusoidalnym kształcie [3]. Klasteryzacji poddawane zostawały także elementy odstające (ang. *outliers*) czy takie, których znaczenie dla danych jest znikome. Przykładowo, gdy chcemy rozpoznawać pismo naturalne, nie mają dużego znaczenia połączenia między literami. Zatem przy założeniu, że generowane przez algorytm centra oddają charakter elementów klastra, rozsądnym wydaje się odrzucanie podciągów, które znacząco nie pasują do żadnego z nich. Między innymi na tej obserwacji opiera się badany przeze mnie algorytm selektywnego grupowania podciągów szeregów czasowych, nazywany w dalszej części pracy **SSTS** (ang. *Selective Subsequence Time Series clustering*).

W mojej implementacji korzystam z dwóch klas, których instancje będą reprezentować odpowiednio podciągi oraz klastry:

```
import numpy as np
class Subsequence:
    def __init__(self, sequence, start, end):
        self.sequence = np.array(sequence, dtype=np.float64)
        self.start = start
        self.end = end
    ...
```

```
class Cluster:
    def __init__(self, s1, s2):
        self.members = [s1, s2]
        self.center = self.average(s1.sequence, s2.sequence)
        self.size = 2

    def average(self, P, Q, w1=1, w2=1):
        return (w1*P + w2*Q)/(w1 + w2)
    ...
```

Rozdział 3.

Analiza algorytmu SSTS

3.1. Konstrukcja bazy danych

Punktem wyjścia dla opisywanej przeze mnie strategii klasteryzacji jest zastosowanie techniki *"sliding window"*. To popularne w dziedzinie szeregów czasowych pojęcie [3] służy do wyodrębnienia z danych wejściowych wszystkich możliwych podciągów, które zebrane w jednej *bazie danych*, stanowić będą przedmiot grupowania. Odbywa się to poprzez przesuwanie ramki o stałej szerokości, która w każdym momencie zawiera w sobie jeden z podciągów. Oczywistym jest, że jej rozmiar determinuje dalszy przebieg algorytmu, zatem będzie on jego głównym parametrem. Nasuwa się wniosek, że ustalanie takiej wartości z góry jest nie tylko niepraktyczne bądź trudne, lecz także uniemożliwia odnajdowanie podobieństw w samej strukturze szeregu czasowego. Podciągi o podobnym kształcie powinny móc znaleźć się w jednej grupie, nawet gdy są różnej długości. Problemem staje się wówczas mierzenie odległości między nimi, jednak autorzy algorytmu SSTS rozwiązują go za pomocą metody ze skalowaniem jednostajnym.

Definicja 5. *Skalowaniem jednostajnym* do długości m (z czynnikiem skalującym $f \geqslant 1$) nazywamy przekształcenie dowolnego z szeregów $T = (t_1, t_2, ..., t_n)$, dla $\lceil m/f \rceil \leqslant n \leqslant \lfloor mf \rfloor$, w wyniku którego otrzymujemy nowy szereg czasowy $T' = (t'_1, t'_2, ..., t'_m)$, gdzie $t'_i = t_{\lceil i \frac{n}{m} \rceil}$.

Oprócz podanej przez użytkownika szerokości *"sliding window"* - w, dodatkowym parametrem staje się liczba rzeczywista $f \geqslant 1$, za pomocą której dysponujemy ramkami o szerokości od $\lceil w/f \rceil$ do $\lfloor wf \rfloor$. W konsekwencji baza danych składa się z podciągów o różnej długości, które za pomocą skalowania jednostajnego rozciągamy albo zwężamy do tej samej długości w.

Przed rozpoczęciem grupowania każdy z podciągów poddawany jest jeszcze *standaryzacji* (ang. *Z-normalization*), dzięki której porównywane będą własności strukturalne a nie konkretne wartości liczbowe [4]. Dodatkowo będziemy mieli możliwość

wykluczenia zbędnych podciągów stałych, dla których odchylenie standardowe wynosi 0.

Definicja 6. *Standaryzacja* szeregu czasowego to przeskalowanie jego elementów na wartości o rozkładzie $\mathcal{N}(0,1)$.
Dany podciąg $T_{i,m} = (t_i, t_{i+1}, ..., t_{i+m-1})$, którego wartością oczekiwaną jest μ, a odchyleniem standardowym σ, po standaryzacji ma postać
$T'_{i,m} = (t'_i, t'_{i+1}, ..., t'_{i+m-1})$, gdzie $t'_k = \frac{t_k - \mu}{\sigma}$.

Wobec powyższych ustaleń dodatkowe procedury oraz funkcja inicjująca algorytm SSTS mają następującą postać:

```
class Subsequence:
    ...
    def uniform_scaling(self, w, f):
        m = len(self.sequence)
        self.sequence = np.array([self.sequence[int(math.ceil(j*(m-1)/(w-1)))]
                                  for j in range(w)])
    def z_normalize(self):
        X = self.sequence
        mu = np.mean(X, axis=0)
        sigma = np.std(X, axis=0)
        if_non_zero = all(map(lambda s: s != float(0), sigma))
        if if_non_zero:
            self.sequence = (X - mu)/sigma
        return if_non_zero

def extract_subsequences(T, w, f):
    S = []
    L = len(T)
    w_min = int(math.ceil(w/f))
    w_max = int(math.floor(w*f))
    for i in range(w_min, w_max+1):
        for j in range(L-i+1):
            s = Subsequence(T[j:j+i], j, j+i)
            s.uniform_scaling(w, f)
            if s.z_normalize():
                S.append(s)
    return S
```

3.2. Kodowanie wejściowego szeregu czasowego

Nasze dążenia do uzyskania optymalnego grupowania wspomagane będą przez ideę kodowania danych zaadaptowaną do charakterystyki szeregów czasowych. Każda z operacji wykonywana przez algorytm może być interpretowana jako konstrukcja lub edycja abstrakcyjnego słownika. Kluczami słownika będą jednoelementowe *symbole kodujące*, a odpowiadającymi wartościami - *wektory kodujące*. Jego zastosowaniem jest zastępowanie elementów z jednego klastra tym samym symbolem kodującym, którego wartością słownikową będzie reprezentacja centroidu. Tym samym słownik wraz z zakodowanym szeregiem czasowym jednoznacznie określa grupowanie. Istotny jest fakt, że realizowane w ten sposób kodowanie możemy traktować jako redukcje długości danych wejściowych, gdyż w kolejnych krokach podciągi są zastępowane przez krótkie symbole. Co więcej, im liczniejszy jest podzbiór podciągów wykorzystanych podczas klasteryzacji, tj. im większa jest kompresja, tym bardziej wartościowy jest rezultat. Wobec tego pierwszym z zasadniczych kryteriów, rozpatrywanych przy wyborze optymalnego stanu wynikowego, jest sumaryczna wartość *wskaźników kompresji* danych.

Definicja 7. *Wskaźnik kompresji* ($\tilde{R}$) jest określony jako stosunek rozmiaru redukcji (uwzględniając konstrukcję słownika) do długości danych wejściowych.

Oczywiście w danych ze świata rzeczywistego bardzo rzadko zaobserwujemy, że każdy klaster będzie składał się z jednakowych podciągów. Wprowadzona zostanie zatem *funkcja błędu*, oceniająca różnice między elementami wewnątrz grupy, która stanowić będzie kolejny składnik miary jakości grupowania.

Definicja 8. Niech C będzie dowolną grupą, a c - jej centrum.
Funkcją błędu nazywamy $E(C) = \sum_{v \in C} dist(c, v)$.

```
class Cluster:
    ...
    def error(self):
        return np.sum([np.linalg.norm(s.sequence - self.center)
                       for s in self.members])
```

Definicja 9. *Błędem grupowania* ($\tilde{E}$) nazywamy sumę wartości funkcji błędu dla każdego z klastrów w grupowaniu.

Intuicja podpowiada, a strategia zastosowana w algorytmie zapewnia, że wraz ze wzrostem kompresji danych rosnąć będzie błąd grupowania. Po analizie głównych procedur opiszemy, jak określić ich optymalny stosunek wyznaczający rozwiązanie.

3.3. Strategia i opis działania

Algorytm iteracyjnie podstawia za bieżący rezultat wartość wynikową jednej z trzech funkcji. Zachłannie wybieramy tę, która minimalizuje *przyrost błędu grupowania* rozumiany jako różnica błędów grupowania po i przed operacją:

$$\Delta\tilde{E} = \tilde{E}_{after} - \tilde{E}_{before}$$

Po każdym obrocie pętli, algorytm SSTS dysponuje błędem grupowania, sumą dotychczasowych współczynników kompresji oraz **bieżącym zbiorem grup**, które są zapamiętywane na liście wszystkich stanów pośrednich - P. Będzie on kontynuował swoje działanie tak długo, dopóki może wykonać *jakąkolwiek* z trzech głównych funkcji, czyli dopóki baza danych nie została opróżniona albo zredukowana do wyłącznie nakładających się podciągów.

```
def SSTS(T, w, f):
    P, cluster_list, compression = [], [], 0
    L = len(T)
    S = extract_subsequences(T, w, f)
    D = np.array([s.sequence for s in S])
    while len(S) or len(cluster_list):
        c_list1, c_list2, c_list3 = map(copy.deepcopy,
                                        repeat(cluster_list, 3))
        s1, s2, s3 = map(copy.deepcopy, repeat(S, 3))
        c, s, d, err, comp = list(zip(create_cluster(c_list1, s1, D, L, w),
                                      update_cluster(c_list2, s2, D, L),
                                      merge_clusters(c_list3, s3, D, L, w)))
        m = np.argmin(err)
        cluster_list, S, D, compression_ratio = c[m], s[m], d[m], comp[m]
        compression += compression_ratio
        if cluster_list:
            P.append([cluster_list, compression,
                      clustering_error(cluster_list)])
    return P, stopping_state(P)


def clustering_error(cluster_list):
    return np.sum([cluster.error() for cluster in cluster_list])
```

3.3.1. Funkcja *Create cluster*

- Zasada działania:
  Tworzy nową grupę z pary najbardziej podobnych podciągów z obecnej bazy danych.

- Wskaźnik kompresji:
  Z dwóch podciągów długości u i v tworzona jest grupa C, której wektorem kodowym zostanie ciąg długości w. Możemy to interpretować jako redukcję danych wejściowych długości L, polegającą na zastąpieniu dwóch podciągów o łącznej długości $(u + v)$ tymi samymi symbolami kodującym długości 1. Tworzymy przy tym wartość słownikową tego symbolu o długości w.

$$\tilde{R} = \frac{(u+v) - (2+w)}{L}$$

- Przyrost błędu grupowania: $\Delta\tilde{E} = E(C)$

Znajdowanie dwóch najbardziej podobnych podciągów odbywa się za pomocą *algorytmu* ***MK*** [2], uznawanego za najefektywniejszy spośród stosujących odległość euklidesową [1]. Po zakończonym wyszukiwaniu konieczne jest usunięcie z bazy danych tej pary oraz wszystkich pozostałych podciągów, które nachodzą na którykolwiek z nich, aby nasz rezultat był zgodny z definicją problemu.

```
def create_cluster(cluster_list, S, D, L, w):
    if len(S) > 1:
        starts = np.array([s.start for s in S])
        i, j = mk_algorithm(D, starts, w, R=6)
        if any([i,j]):
            C = Cluster(S[i], S[j])
            inc_error = C.error()
            u = S[i].end - S[i].start
            v = S[j].end - S[j].start
            comp_ratio = (u + v - (2 + w))/L
            cluster_list.append(C)
            S, D = remove_subsequences(S, D, [i, j])
            return cluster_list, S, D, inc_error, comp_ratio
    return [], S, D, np.inf, 0
```

3.3.2. Funkcja *Update cluster*

- Zasada działania:
  Do jednej z grup dodaje podciąg najbardziej podobny do jej centroidu.
- Wskaźnik kompresji:
  Podciąg długości u dodajemy do jednej z aktualnie istniejących grup C i wyliczamy na nowo jej wektor kodowy za pomocą średniej ważonej u oraz centrum C, z wagami odpowiednio 1 oraz $|C|$. W ten sposób powstaje nowy klaster C'. W miejsce podciągu umieszczamy symbol kodowy długości 1 i nie zmieniamy przy tym rozmiaru aktualnego słownika.
  $$\tilde{R} = \frac{u-1}{L}$$
- Przyrost błędu grupowania: $\Delta\tilde{E} = E(C') - E(C)$

Wybór klastra i podciągu przebiega następująco: dla każdej grupy, z bazy danych wyszukiwany jest podciąg najbardziej podobny do jej centroidu. Dodawany jest wtedy, gdy skutkuje to najmniejszą wartością funkcji błędu dla każdej z grup. Podobnie jak w przypadku *Create cluster* - z bazy danych musimy usunąć wszystkie podciągi nachodzące na ten wybrany.

```
def update_cluster(cluster_list, S, D, L):
    if len(cluster_list) and len(S):
        j, min_err = None, np.inf
        for i in range(len(cluster_list)):
            C = copy.deepcopy(cluster_list[i])
            t = subsequence_matching(S, D, C)
            if t is None:
                continue
            C.add_member(S[t])
            err = C.error()
            if err < min_err:
                j, min_err, updated_C, added_t = i, err, C, t
        if j is None:
            return [], S, D, np.inf, 0
        inc_error = min_err - cluster_list[j].error()
        u = S[added_t].end - S[added_t].start
        comp_ratio = (u - 1)/L
        cluster_list[j] = updated_C
        S, D = remove_subsequences(S, D, [added_t])
        return cluster_list, S, D, inc_error, comp_ratio
    return [], S, D, np.inf, 0
```

3.3.3. Funkcja *Merge clusters*

- Zasada działania:
  Scala dwie najbardziej podobne grupy.

- Wskaźnik kompresji:
  Łącząc dwie grupy C_1 i C_2 sumujemy zbiory ich członków, a centroidem nowo powstałego klastra C' jest uśredniona wartość dwóch poprzednich z wagami odpowiednio $|C_1|$ oraz $|C_2|$. Ze słownika usuwamy dwa wektory kodowe i dodajemy jeden nowy, wszystkie o długości w.

$$\tilde{R} = \frac{w}{L}$$

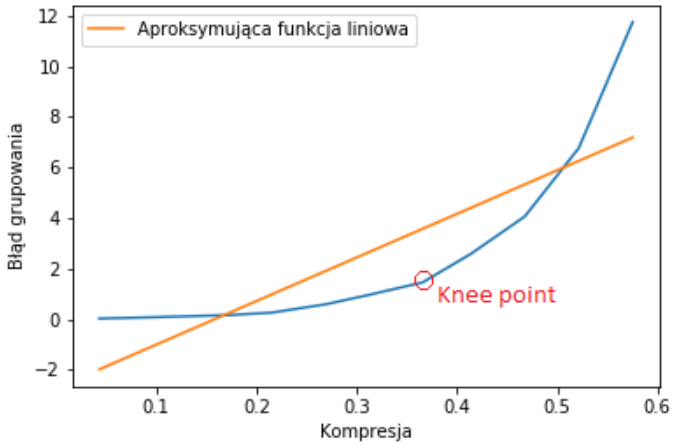
- Przyrost błędu grupowania: $\Delta\tilde{E} = E(C') - (E(C_1) + E(C_2))$

Stosujemy tu rozwiązanie siłowe ze względu na niewielką liczbę grup w dowolnym momencie działania programu. Scalamy wszystkie możliwe pary grup i za rezultat przyjmujemy połączenie, w wyniku którego otrzymamy minimalną wartość funkcji błędu dla nowo powstałej grupy.

```
def merge_clusters(cluster_list, S, D, L, w):
    n = len(cluster_list)
    if n > 1:
        min_err = np.inf
        for i in range(n-1):
            for j in range(i+1, n):
                C1 = copy.deepcopy(cluster_list[i])
                C2 = copy.deepcopy(cluster_list[j])
                C1.merge(C2)
                err = C1.error()
                if err < min_err:
                    merged_C = C1
                    l1, l2 = i, j
                    min_err = err
        inc_error = min_err - (cluster_list[l1].error() + cluster_list[l2].error())
        comp_ratio = w/L
        cluster_list[l1] = merged_C
        del cluster_list[l2]
        return cluster_list, S, D, inc_error, comp_ratio
    return [], S, D, np.inf, 0
```

3.4. Stan wynikowy

Istotna jest obserwacja, że od pewnego momentu klasteryzacje istotnie tracą na jakości ze względu na generowany duży błąd grupowania. Spowodowany jest on wykonywaniem kolejnych operacji niejako na siłę, gdyż algorytm jest zmuszony wywołać np. funkcję *Update cluster*, w której wybrany podciąg nie pasuje już tak dobrze do wskazanej grupy. Nic też nie stoi na przeszkodzie, by w końcowych iteracjach SSTS scalał dwa klastry, które nie powinny być scalone. Wobec tego istnieje stan wynikowy, w którym relacja błędu grupowania do uzyskanej kompresji jest optymalna, a po którym błąd zaczyna drastycznie rosnąć. Wskazać go można za pomocą funkcji obrazującej ową relację. Mianowicie dzięki aproksymacji jej wielomianem pierwszego stopnia, jesteśmy w stanie wskazać *"knee point"* - punkt, w którym różnica między wartościami na prostej a wartościami przybliżanej funkcji jest największa.



Rysunek 3.1: Przykładowa ilustracja lokalizacji *"knee point"*.

Oczywiście ten wykres, a w konsekwencji wybór rozwiązania spośród wszystkich stanów pośrednich, możemy sporządzić dopiero po zakończeniu działania programu.

```
def stopping_state(P):
    data = np.array(P)[:,1:]
    X = np.array(data[:, 0], dtype=np.float64)
    Y = np.array(data[:, 1], dtype=np.float64)
    a, b = np.polyfit(X, Y, 1)
    residuas = [((a * X[i] + b) - Y[i]) for i in range(len(X))]
    return np.argmax(residuas)
```

Rozdział 4.

Wariant dla danych wielowymiarowych

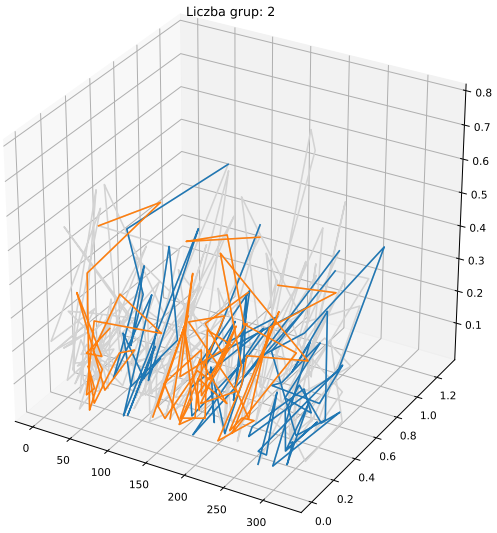
W swojej implementacji rozszerzyłem algorytm SSTS o możliwość działania na wielowymiarowych szeregach czasowych. Podobnie jak w wariancie jednowymiarowym, utożsamiamy je z ich reprezentacją.

Definicja 10. *Szeregiem czasowym wielowymiarowym* T długości m nazywamy ciąg wektorów $v_i \in \mathbb{R}^n$, $n \geqslant 2$, taki że $T = (v_1, v_2, ..., v_m)$. Składa się on z n jednowymiarowych szeregów czasowych T_i, takich że $t_{ij} = v_{ji}$.

Definicja podciągu wielowymiarowego szeregu czasowego jest analogiczna. Przypadek wielowymiarowy zmusza nas jednak do dostosowania metody obliczania odległości między podciągami, będącej składową każdej z procedur. Obecnie zamiast ciągów liczb mamy ciągi wektorów, więc podciągi szeregu czasowego mają postać macierzy. Korzystamy zatem z uogólnienia metryki euklidesowej zdefiniowanej jako wartość *normy Frobeniusa* dla różnicy macierzy:

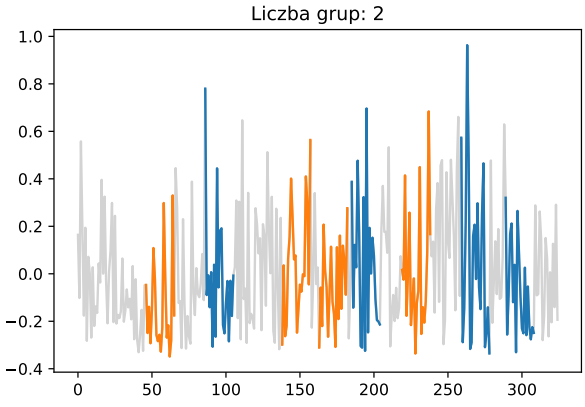
$$dist(A, B) = ||A - B||_F = \sqrt{\sum_{i,j}(a_{ij} - b_{ij})^2}$$

Do większej liczby wymiarów dostosować należy także metodę standaryzacji. Dla danego podciągu będziemy ją wykonywać osobno dla każdego wymiaru tak, jakbyśmy standaryzowali składowe jednowymiarowe szeregi czasowe. Pozostała część algorytmu zasadniczo pozostaje niezmieniona. Pojawia się natomiast problem z wizualizacją otrzymanych wyników, gdyż nie jesteśmy w stanie z łatwością sporządzić wykresów dla szeregów czasowych o wymiarowości większej niż 2. Dla samych dwuwymiarowych danych ze świata rzeczywistego owszem istnieją przypadki, w których wykres rezultatu zawiera cenne informacje. W rozdziale 6. pokażemy przykładowe wyniki takich doświadczeń. Często jednak wizualizacje są bardzo nieczytelne:



Rysunek 4.1: Przykładowa wizualizacja rezultatu dla danych dwuwymiarowych [6].

W związku z tym postanowiłem do samej wizualizacji wyników zastosować metodę analizy składowych głównych - *PCA* (ang. *Principal Component Analysis*). Polega ona takim obrocie układu współrzędnych, aby wartości wariancji na współrzędnych były malejące względem ich kolejności. Dzięki temu pierwsza współrzędna, określana jako pierwsza składowa główna, wyjaśnia najwięcej zmienności w danych. Wartości z tej współrzędnej traktuję jako dane wielowymiarowe zredukowane do jednego wymiaru. Niniejszym algorytm grupujący wykonywany jest na pełnych danych, a otrzymane wyniki przenosimy na jednowymiarowy szereg czasowy uzyskany dzięki redukcji wymiarowości.



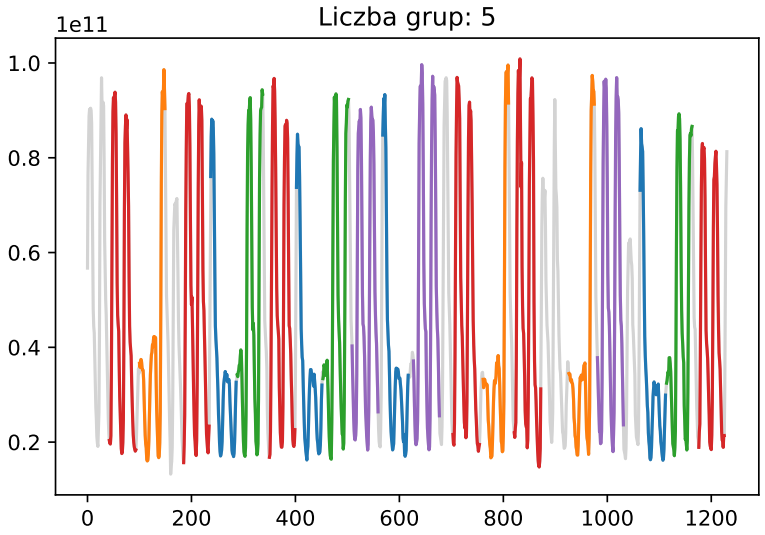
Rysunek 4.2: Wizualizacja rezultatu dla tych samych danych przy pomocy PCA.

Rozdział 5.

Wyniki doświadczeń

5.1. Dane jednowymiarowe

Zgodnie z zapewnieniami autorów, algorytm SSTS produkuje zadowalające wyniki dla danych jednowymiarowych. Wskazywane przez niego grupy *zawsze* opisują faktyczne podobieństwa między podciągami, generując znaczące wektory kodowe. Zwracają także uwagę na subtelne różnice we fragmentach, które mogłyby zostać uznane przez człowieka za niemal identyczne.

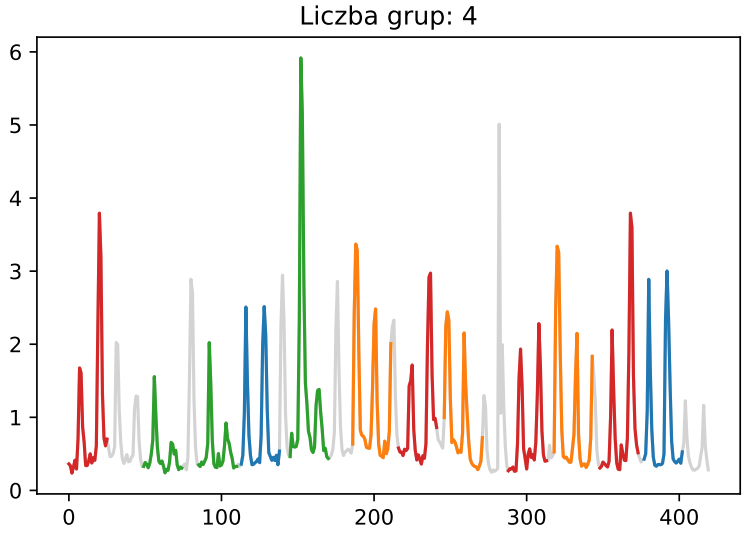


Rysunek 5.1: Rezultat dla danych [7].
$w = 60$, $f = 1.2$

Powyższa klasteryzacja ilustruje wychwytywanie różnic w podciągach, polegających na kształtach ich zakończeń. Grupa fioletowa oraz czerwona są niemal identyczne, lecz wszystkie fioletowe podciągi zaczynają się szpicem w dół. Ponadto, w

grupie oznaczonej kolorem czerwonym pierwszy górny szpic jest zawsze wyższy od drugiego, podczas gdy fioletowe są bardziej zbliżone do siebie. Dodatkowo można wyciągać wnioski z kolejności występowania elementów różnych grup. Przykładowo każdy podciąg zaznaczony na powyższym wykresie kolorem zielonym jest poprzedzany przez podciąg niebieski.

Natomiast w poniższym rezultacie na podstawie grupy zielonej doskonale widać wychwytywanie samych podobieństw strukturalnych, dzięki wykonanej standaryzacji.

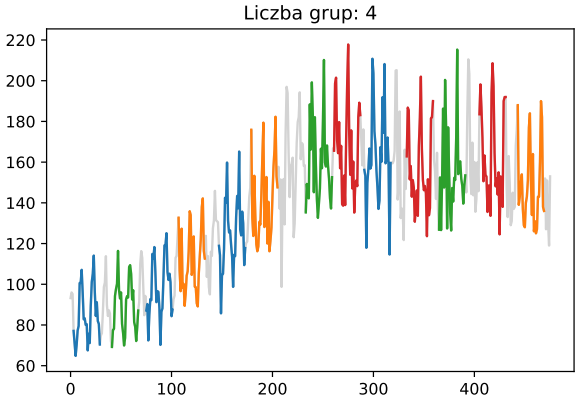


Rysunek 5.2: Rezultat dla danych [8].
$w = 26$, $f = 1$

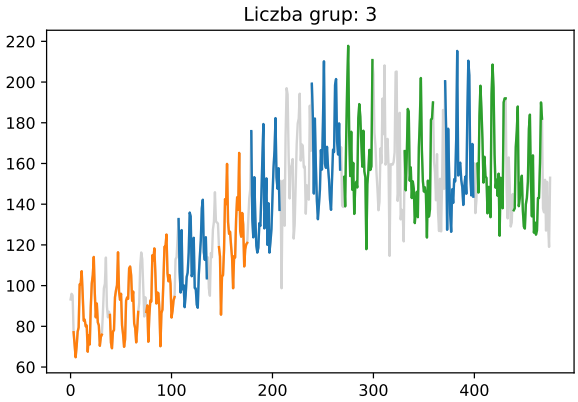
Niestety algorytm ma też swoje wady. Główną z nich jest jego czas działania dla czynników skalujących $f > 1$. Takie wartości powodują względnie duży rozmiar bazy danych. Przykładowo dla wejściowego szeregu czasowego długości 1000, *"sliding window"* rozmiaru 50 oraz $f = 1$, baza danych ma rozmiar 951. Rozwiązanie zostanie znalezione w mniej niż 7 sekund. Natomiast gdy czynnik skalujący zmienimy na 1.5, dysponujemy aż 39753 podciągami. Grupowanie będzie dokładniejsze, gdyż wydobyte zostaną podciągi o długości od 34 do 75. Jednak złożoność czasowa SSTS jest zdominowana przez złożoność *algorytmu MK*, którego zasada działania nie jest dostosowana dla tak uzyskanych danych. Samo znalezienie pary najbardziej podobnych podciągów w pierwszej iteracji trwa niemal 5 minut. Dzieje się tak, gdyż wówczas dużo więcej podciągów nachodzi na siebie oraz wiele elementów z bazy danych to w istocie niemalże *te same* podciągi, nieznacznie przesunięte lub przeskalowane względem siebie.

Drugą z wad jest konieczność ustalania z góry dwóch parametrów wejściowych: w oraz f. Predefiniowany rozmiar ramki w połączeniu z nietrywialnym czynnikiem

skalującym zapewniają szeroki zakres długości grupowanych podciągów. Miało to na celu, oprócz umożliwienia klasteryzacji podciągów o różnych długościach, zniwelowanie potrzeby podania dokładnych wartości przez użytkownika. Jednak rozwiązania okazują się wrażliwe na niewielkie zmiany tych argumentów.



Rysunek 5.3: Grupowanie danych [9].
$w = 27,\ f = 1$



Rysunek 5.4: Grupowanie danych [9].
$w = 29,\ f = 1$

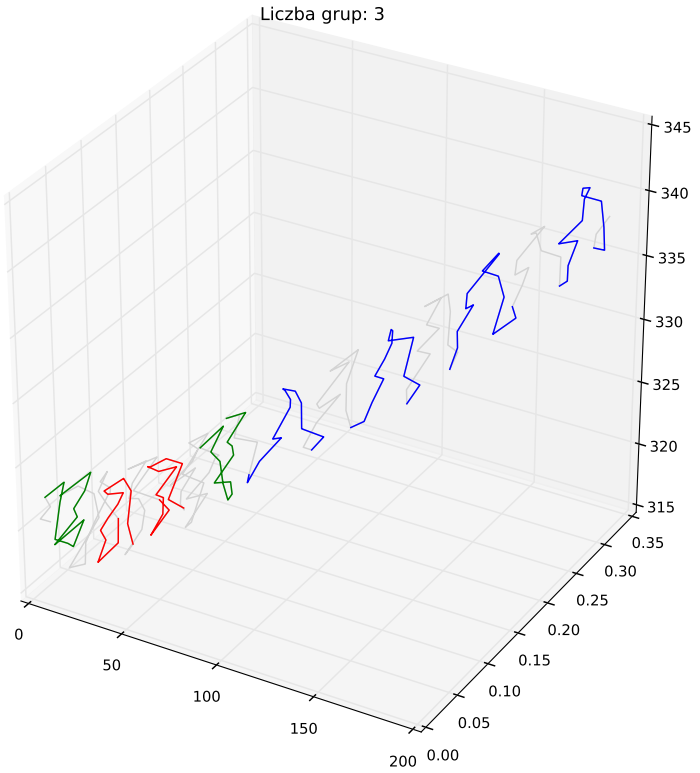
Oczywiście po chwilowym wglądzie w dane oraz po kilku eksperymentach, rozwiązania będą spełniać nasze oczekiwania. Ostatecznie — jeśli wśród danych da się zaobserwować podobne podciągi, tj. jeśli dane *nadają się* do grupowania, to algorytm SSTS efektywnie je pogrupuje.

5.2. Dane wielowymiarowe

Okazuje się, że ten algorytm znajduje swoje zastosowanie także przy danych wielowymiarowych, ponieważ w tym przypadku wynikowe klasteryzacje również wskazują na zbiory podobnych podciągów. Niestety w tym wariancie jest dużo ciężej o

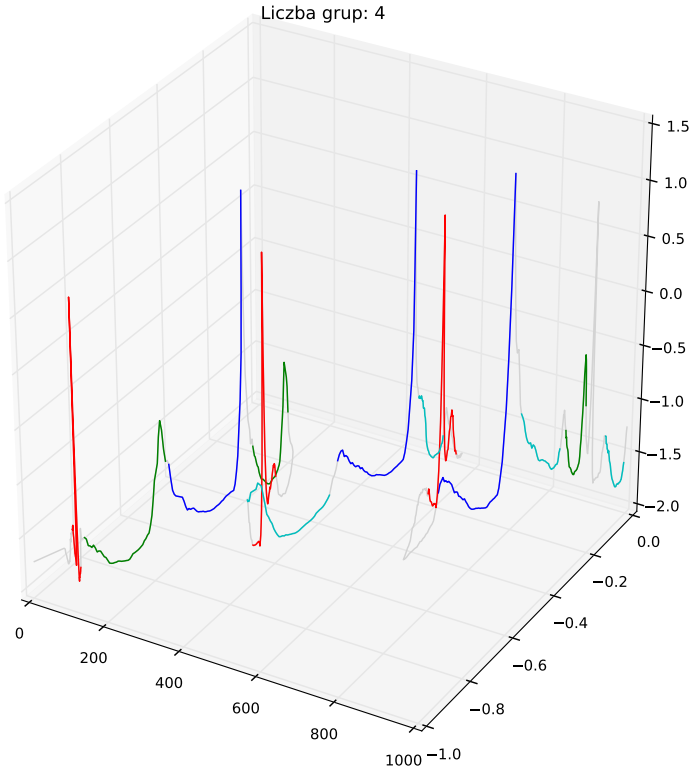
te podobieństwa, gdyż często nie ma bezpośredniej zależności między składowymi szeregami czasowymi. Z tego powodu odległości między podciągami bywają na tyle duże, że przy względnie większej szerokości ramki wszystkie podciągi zostają przyporządkowane do jednej grupy.

Jednak istnieją dane, dla których użyteczność metody jest zauważalna. Przykładowo, dla dwuwymiarowego szeregu czasowego opisującego emisje CO_2 algorytm dokładnie wskazuje na podobne podciągi, które z łatwością można dostrzec na pełnym wykresie, bez konieczności stosowania technik redukcji wymiarowości:



Rysunek 5.5: Grupowanie dla danych [10],
$w = 14,\ f = 1.1$

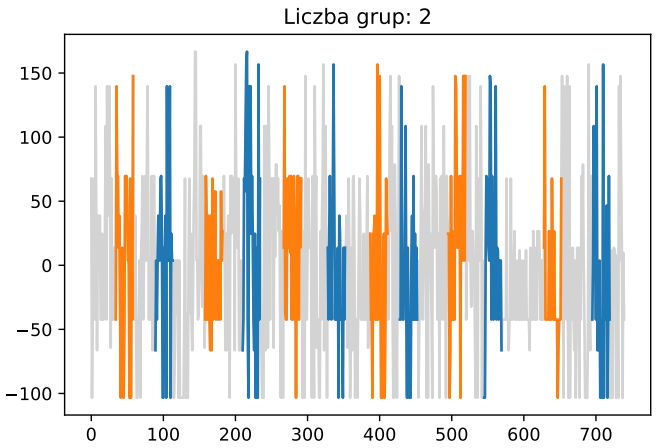
Dodatkowo podczas kolejnych eksperymentów na sztucznie spreparowanych danych dwuwymiarowych okazało się, że wyniki algorytmu potrafią być tak samo dobre, jak w przypadku jednowymiarowych szeregów czasowych. Na poniższym przykładzie widać, że algorytm wykonał grupowanie bardzo trafnie, gdyż każda z grup składa się z podciągów o niemal jednakowej strukturze:



Rysunek 5.6: Grupowanie dla sztucznie spreparowanych danych, $w = 50,\ f = 1$

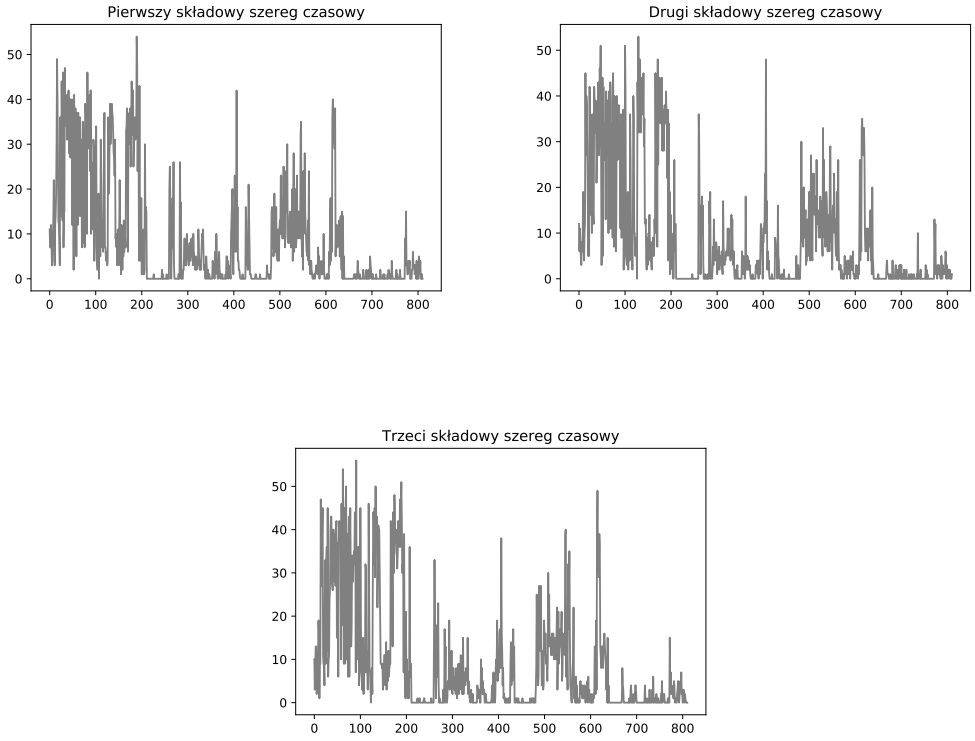
O tak zadowalającą czytelność wyników zazwyczaj trudno. Wraz ze wzrostem wymiarowości coraz ciężej o wychwytywanie podobieństw w podciągach, gdyż każda dodatkowa cecha wprowadza kolejne zaburzenia. Na przykład dla danych trójwymiarowych podciągi mogą być bardzo zbliżone na dwóch pierwszych współrzędnych, podczas gdy wartości na trzeciej współrzędnej zwiększą odległość między podciągami na tyle, że algorytm nie uzna ich za podobne. Z tego powodu SSTS w większości przypadków może co najwyżej służyć do naprowadzania na pewne podobieństwa, które później należałoby zweryfikować i w inny sposób ocenić ich znaczenie.

Poniżej znajduje się rezultat grupowania dla trójwymiarowego szeregu czasowego, określającego okoliczności absencji pracowników w pewnej firmie w ciągu 3 lat. Uzyskaliśmy zaledwie dwie grupy, o których możemy powiedzieć jedynie tyle, że występują naprzemiennie:



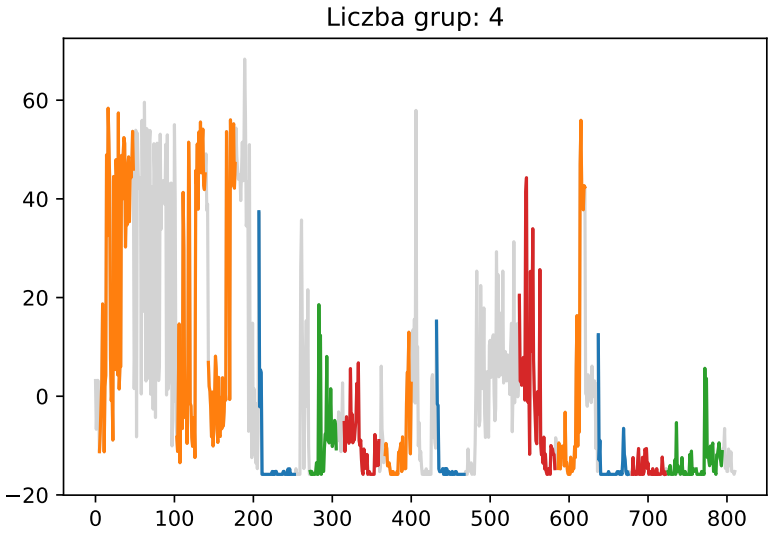
Rysunek 5.7: Wynik grupowania dla trójwymiarowych danych [11], $w = 25,\ f = 1$

Jako ostatni przykład rozpatrzymy trójwymiarowy szereg czasowy opisujący tygodniowe wyniki sprzedaży trzech wybranych produktów w pewnej sieci sklepów na przestrzeni kilku lat. Okazuje się, że wśród tych danych występuje pewna regularność, dzięki której wyniki klasteryzacji zawierają dużo więcej informacji. Poniżej znajdują się wykresy trzech składowych jednowymiarowych szeregów czasowych, z których widać, że podobieństwa między nimi są bardzo znaczące.



Rysunek 5.8: Wykresy składowych trójwymiarowego szeregu czasowego [12].

Grupowanie zilustrowane na wykresie otrzymanym za pomocą metody *PCA* jest o tyle wartościowe, że z łatwością możemy zweryfikować jego trafność. Jesteśmy w stanie wyróżnić grupy reprezentujące wzrosty wartości: pomarańczową — podciągi o większym wzroście oraz zieloną — podciągi o mniejszym wzroście, po którym następuje spadek. Podobnie grupy opisujące spadki wartości: niebieska — gwałtowny spadek do wartości minimalnej, czerwona — spadek poprzedzony wahaniami.



Rysunek 5.9: Wynik grupowania dla trójwymiarowych danych [12], $w = 40$, $f = 1.2$

Zatem nasuwa się wniosek, że zastosowanie algorytmu SSTS dla wielowymiarowych szeregów czasowych wraz z tą metodą wizualizacji ma największy sens wtedy, gdy chcemy wyciągnąć zbiorczy wniosek z danych, wśród których spodziewamy się podobieństw pomiędzy prawie wszystkimi cechami na pewnych podciągach. Dzięki takim rezultatom możemy formułować ogólniejsze predykcje oraz charakteryzować pewne przedziały czasowe.

5.3. Wartości liczbowe

W jednej tabeli zebrałem liczby opisujące wyniki części z przeprowadzonych eksperymentów. Informują nas one o różnicach w czasie wykonywania algorytmu, w zależności od podanych parametrów, oraz o osiągniętych przez grupowanie wartościach. Należy zwrócić uwagę na wielokrotny wzrost czasu działania przy $f > 1$, dla którego jednak w większości przypadków uzyskiwana jest większa liczba grup.

Dane	Rozmiar	w	f	Czas działania	Liczba grup	Kompresja	Błąd grupowania
[13]	2820 x 1	150	1	74.59 s	4	0.52695	56.64626
[13]	2820 x 1	150	1.2	63.51 h **[!]**	5	0.56986	65.85948
[8]	420 x 1	26	1	4.77 s	4	0.46667	12.14659
[8]	420 x 1	26	1.5	7.52 min	5	0.35714	9.55411
[7]	1231 x 1	60	1	5.81 s	4	0.52396	8.90239
[7]	1231 x 1	60	1.2	14.53 min	5	0.51990	11.74186
[6]	325 x 6	10	1	16.71 s	2	0.32615	81.10944
[6]	325 x 6	10	1.5	13.49 min	1	0.073846	19.67943
[12]	811 x 3	40	1	35.38 s	3	0.429	67.49368
[12]	811 x 3	40	1.2	1.57 h	4	0.45376	74.33416

Rozdział 6.

Podsumowanie

Algorytm zaimplementowany na podstawie pracy [1] okazał się skutecznym narzędziem do grupowania podciągów szeregów czasowych. Zastosowana strategia, w której odrzucamy pewne najmniej pasujące podciągi, skutkuje powstawaniem grup odzwierciedlających dokładne podobieństwa wśród danych. Dodatkowo użycie standaryzacji i techniki ze skalowaniem umożliwia wskazywanie podobieństw strukturalnych niezależnych od wartości liczbowych. Algorytm SSTS nadaje się również do rozszerzenia o możliwość działania na danych wielowymiarowych. Wówczas przy pewnych dodatkowych założeniach dotyczących danych produkuje on znaczące rezultaty, które jesteśmy w stanie wizualizować.

W kwestii przyszłego rozwoju algorytmu sądzę, że dopracowania wymaga metoda znajdowania pary najbardziej podobnych podciągów. Potrzebna jest taka, która bardziej pasowałaby do charakterystyki otrzymanej bazy danych, gdyż algorytm MK marnuje dużo czasu na porównywanie odległości między nachodzącymi na siebie podciągami. Obecnie użyteczność zastosowania czynnika skalującego może być podważalna właśnie ze względu na bardzo długi czas działania programu.

Bibliografia

[1] S. Rodpongpun, V. Niennattrakul, C. A. Ratanamahatana, *Selective Subsequence Time Series clustering*, Knowledge-Based Systems 35 (2012) 361–368.

[2] A. Mueen, E.J. Keogh, Q. Zhu, S. Cash, B. Westover, *Exact Discovery of Time Series Motifs*, in: SIAM International Conference on Data Mining (SDM'09), Nevada, USA, 2009, pp. 473–484.

[3] S. Zolhavarieh, S. Aghabozorgi, Y. W. Teh, *A Review of Subsequence Time Series Clustering*, Hindawi Publishing Corporation, The Scientific World Journal, Volume 2014.

[4] P. Senin, *Z-normalization of Time Series*, SAX-VSM, URL: `https://jmotif.github.io/sax-vsm_site/morea/algorithm/znorm.html` (31.08.2018)

[5] Monthly New York City births Dataset, URL: `http://bit.ly/WZ9ca0` (31.08.2018)

[6] D. Meko, Six tree-ring sites Dataset, URL: `http://bit.ly/2wrMlc8` (31.08.2018)

[7] Internet traffic Dataset, URL: `http://bit.ly/1StSQ4k` (31.08.2018)

[8] Rock Creek Dataset, URL: `http://bit.ly/2PRZ6Vh` (31.08.2018)

[9] Monthly beer production Dataset, URL: `http://bit.ly/1hwDU5g` (31.08.2018)

[10] CO_2 (ppm) mauna loa Dataset, URL: `http://bit.ly/1qP3iF0` (31.08.2018)

[11] Absenteeism at work Dataset, URL: `https://archive.ics.uci.edu/ml/datasets/Absenteeism+at+work` (31.08.2018)

[12] Weekly Sales Transactions Dataset, URL: `https://www.kaggle.com/crawford/weekly-sales-transactions/version/1#` (31.08.2018)

[13] Zuerich monthly sunspot numbers Dataset, URL: `http://bit.ly/1Dl1ZUb` (31.08.2018)