POLITECHNIKA POZNAŃSKA

WYDZIAŁ INFORMATYKI I TELEKOMUNIKACJI
Instytut Informatyki

Praca dyplomowa magisterska

OPRACOWANIE I IMPLEMENTACJA METODY OCENY PODOBIEŃSTWA SZEREGÓW CZASOWYCH, REPREZENTUJĄCYCH ZUŻYCIE ZASOBÓW KOMPUTEROWYCH PRZEZ PROCESY PRZETWARZANIA DANYCH

Anna Lehnhardt, 136275
Bartosz Ciesielski, 136694

Promotor
dr hab. inż. Robert Wrembel, prof. nadzw.
mgr inż. Michał Bodziony (senior soft. arch. IBM Software Lab, Kraków)

POZNAŃ 2022

Streszczenie

Celem pracy jest znalezienie efektywnych metod określenia operatorów wykorzystanych w danym UDF (User Defined Function) na podstawie szeregów czasowych reprezentujących zużycie zasobów komputerowych. Analizowane operatory to filtracja, agregacja i projekcja oraz ich kombinacje.

Dane z szeregami czasowymi przedstawiającymi zużycie CPU oraz pamięci RAM otrzymano od autorów pracy magisterskiej z 2021 roku: „Odkrywanie charakterystyk User Defined Functions". Zawierały one szeregi powstałe przy przetwarzaniu danych o rozmiarze 1GB oraz 2GB.

W ramach pracy wykonano eksperymenty wykorzystujące algorytmy badające podobieństwo szeregów czasowych (odległość euklidesowa, Dynamic Time Warping, Longest Common Subsequence) oraz algorytmy klasyfikacji wykorzystujące uczenie maszynowe (ROCKET, HIVECOTE2, KNN).

Na podstawie przeprowadzonych badań można wyciągnąć następujące wnioski: uczenie maszynowe, a dokładnie algorytm ROCKET jest najbardziej efektywną z przetestowanych metod. Umożliwiła ona bardzo dokładne rozróżnienie typów funkcji UDF.

Abstract

The aim of this thesis is to find effective methods of determining the operations used in a given UDF (User Defined Function) based on the time series data representing the usage of computer resources. The analyzed operations are filtering, aggregation, and projection, and their combinations.

Time series datasets showing the consumption of CPU and RAM were obtained from the authors of the thesis from 2021: "Discovering User Defined Functions". Results of that paper contained time series data from the processing of dataset that were of 1 GB and 2 GB of size.

As part of the work, experiments were carried out using algorithms examining the similarity of time series (Euclidean distance, Dynamic Time Warping, Longest Common Subsequence) and classification algorithms using machine learning (ROCKET, HIVECOTE2, KNN).

Based on the research, the following conclusions can be drawn: machine learning, and more precisely, the ROCKET algorithm is the most effective of the tested methods. That algorithm made it possible to distinguish the types of UDFs very accurately.

Spis treści

1 Wstęp **1**

2 Przegląd badań **3**

2.1 Podobieństwo szeregów czasowych - przegląd algorytmów . . . 3

2.1.1 Odległość Euklidesowa . . . 4

2.1.2 Dynamiczna transformata czasowa (Dynamic Time Warping) . . . 4

2.1.3 Najdłuższy wspólny podciąg (Longest Common Subsequence) . . . 5

2.1.4 Landmark Similarity . . . 5

2.2 Klasyfikacja szeregów czasowych . . . 6

2.2.1 Oceny jakości klasyfikatorów . . . 6

2.2.2 ROCKET . . . 7

2.2.3 HIVECOTE2 . . . 8

2.2.4 KNN-DTW . . . 10

2.3 Metody normalizacji danych . . . 11

2.3.1 Min-Max . . . 11

2.3.2 Z-score . . . 11

3 Opis danych **12**

3.1 Struktura i schemat danych surowych . . . 12

3.2 Charakterystyka danych wejściowych . . . 14

3.3 Charakterystyka danych w kontekście wyboru algorytmu . . . 21

4 Oprogramowanie i przygotowanie danych **22**

4.1 Wykorzystane biblioteki do eksperymentów . . . 22

4.1.1 Biblioteki implementujące algorytmy obliczające podobieństwo szeregów czasowych . . . 22

4.1.2 Biblioteki implementujące algorytmy klasyfikujące szeregi czasowe . . . 24

4.2 Przetwarzanie i wizualizacja danych . . . 26

4.2.1 Przepływ przetwarzania danych . . . 26

4.2.2 Uporządkowanie danych . . . 27

4.2.3 Normalizowanie danych . . . 31

4.2.4 Wygładzenie danych . . . 32

4.2.5 Przygotowanie zbioru danych eksperymentalnych . . . 33

5 Eksperymenty **36**

5.1 Podobieństwo typów funkcji . . . 36

5.1.1 Algorytmy podobieństwa szeregów czasowych . . . 36

5.1.2 Algorytmy uczenia maszynowego . . . 57

5.2 Podobieństwo wolumenów danych 68
5.2.1 Algorytmy badające podobieństwo 68
5.2.2 Algorytmy uczenia maszynowego 73

6 Podsumowanie eksperymentów **88**
6.1 Podobieństwo szeregów 88
6.2 Uczenie maszynowe 89
6.3 Porównanie podejść 90
6.3.1 Porównanie eksperymentów badających podobieństwo typu funkcji 90
6.3.2 Porównanie eksperymentów badających podobieństwo rozmiarów przetwarzanych danych 91

7 Wnioski i perspektywy badawcze **92**

Literatura **95**

A Repozytorium **98**

Rozdział 1

Wstęp

W obecnym przemyśle i środowisku naukowym co raz częściej konieczne jest przetwarzanie dużych wolumenów danych (tzw. Big Data). Analiza takich danych może wymagać złożonych rozwiązań, które nie wykorzystują wbudowanych metod. Umożliwione to jest poprzez wykorzystanie UDF (ang. *User Defined Functions*), udostępnione jako komponenty zapytań SQL czy w to co raz popularniejszym framework'u Spark. Operacje wykonywane wewnątrz takich funkcji są nieznane systemowi, przez co system nie jest wstanie wykorzystać mechanizmów optymalizujących zapytania. Powoduje to nieefektywne przetwarzanie i niską wydajność systemów przetwarzających dane. Potencjalnym rozwiązaniem, które dałoby możliwość przyspieszenia procesowania UDF'ów, byłoby wykrycie typów operacji wewnątrz tych funkcji i wykorzystanie tej wiedzy w celu zoptymalizowania zapytania.

Świat naukowy podejmuje próbę znalezienia sposobu identyfikacji uruchamianych aplikacji, jednak raczej nie skupiają się na zbadaniu ich zawartości. Głównym celem [1] było poprawienie estymat długości przetwarzania się danych, określanych przez użytkowników systemu. W tym przypadku wykorzystane zostały historyczne metadane uruchomień przez osoby uruchamiające aplikacje. Podobnie, praca [2] również poprawiała estymaty, jednak w jej przypadku wykorzystywano charakterystyki uruchamianych historycznie aplikacji. Autorzy artykułu [3] podjęli próbę optymalizacji przetwarzania zadań w technologii Spark. Tutaj jednak skupiono się na optymalizacji parametrów, nadawanych na potrzeby przetwarzania silnikowi Spark, zamiast badać bezpośrednio działanie aplikacji. Pokazuje to, że tematyka badania aplikacji przetwarzających dane typu Big Data jest wciąż bardzo popularna i innowacje w tej dziedzinę są potrzebne.

Efektem pracy [4] było utworzenie zbiorów danych, które można podzielić na przetwarzające różne typy wykonywanych operacji (agregacja, filtracja, połączenia oraz ich kombinacje). Przedstawiają one szeregi czasowe, które reprezentują zużycie zasobów przez procesy przetwarzania danych. Przy ich pomocy autorzy [4] stwierdzili, że występuje pewne podobieństwo między funkcjami z tej samej grupy. Pokazuje to, że sensowne jest dalsze zbadanie tych danych próbując znaleźć inne metody, które wykazywałby istnienie tego podobieństwa. Dodatkowo praca pilotażowa [5] zaproponowała metody, które pozwalają na odkrywanie semantyki prostych UDF. Podejście to pozwala rozpoznać operatory takie jak: agregacja, filtracja, projekcja czy filtracja z projekcją. Oznacza to, że jest możliwe rozpoznanie tychże operacji i zbadanie innych metod.

Tym samym poniższa praca podejmuje próbę znalezienia najefektywniejszych metod, które umożliwiłby **określenie typów funkcji na podstawie szeregów czasowych reprezentujących zużycie zasobów komputerowych przez procesy przetwarzania danych**. Wykorzystywane w tym celu zostały **algorytmy badające podobieństwo tychże szeregów oraz klasyfikacja przy pomocy uczenia maszynowego.** Wykorzystane algorytmy badają podo-

bieństwo dla przetwarzanych danych **o różnych rozmiarach i zbadano, czy rozmiar utrudnia rozróżnienie operacji tego samego typu.**

Struktura pracy jest następująca. W rozdziale 2. opisane zostały algorytmy wykorzystywane w pracy. Dotyczą one podobieństwa szeregów oraz uczenia maszynowego. W Rozdziale 3. zbadano i opisano charakterystykę przetwarzanych danych przed ich wstępnym przetwarzaniem. Następnie w rozdziale 4. przedstawiono wybrane technologie oraz biblioteki implementujące wybrane algorytmy. Dodatkowo w dalszej części, pokazano przepływ przetwarzania danych oraz opisano wykonane operacje przygotowujące dane dla wybranych bibliotek. Rozdział 5. zawiera wyniki wykonanych eksperymentów w ramach niniejszej pracy. Rozdział 6 opisuje wyniki uzyskane w poprzednim rozdziale, oraz porównuje zbadane podejścia. Ostatni rozdział 7. stanowi podsumowanie pracy. Zawarto w nim wnioski oraz perspektywy dalsze perspektywy badawcze.

Podział zadań wykonywanych ramach niniejszej pracy.

Bartosz Ciesielski odpowiadał za:

- wstępne przetwarzanie danych,
- analizę algorytmów uczenia maszynowego pod kątem wykorzystania ich z szeregami czasowymi,
- zbadanie bibliotek języka Python zawierających algorytmy uczenia maszynowego,
- zaimplementowanie oraz przeprowadzenie eksperymentów wykorzystujących uczenie maszynowe,
- analizę oraz wizualizację wyników związanych z uczeniem maszynowym.

Anna Lehnhardt odpowiadała za:

- opis charakterystyki danych wejściowych,
- analizę algorytmów obliczających podobieństwo szeregów czasowych,
- wybór bibliotek implementujących algorytmy podobieństwa szeregów czasowych,
- implementację i przeprowadzenie eksperymentów z wykorzystaniem algorytmów obliczających podobieństwo,
- przedstawienie wyników eksperymentów, których celem było obliczenie podobieństwa szeregów czasowych.

Rozdział 2

Przegląd badań

W literaturze można znaleźć prace naukowe, które badają sposoby identyfikowania aplikacji uruchamianych w środowisku superkomputerowym. Często jednak określenie typu wykonywanego programu nie jest głównych celem tych artykułów. W pracy [1] autorzy próbowali poprawić estymacje planowanych zadań . Do identyfikacji wykorzystali metadane historyczne w celu znalezienia przeszłych uruchomień aplikacji przez danego użytkownika w ramach projektu. W zbiorach danych wykorzystywanych w pracy nie ma takich informacji, dlatego nie może zostać zastosowana podobna metoda.

W [2], gdzie autorzy szukali metody poprawienia estymat długości czasu przetwarzania się aplikacji w systemie. Chcieli oni znaleźć metody klasyfikacji typu aplikacji bez wykorzystywania metadanych. Zrobili to klasyfikując aplikacje na podstawie charakterystyk historycznych uruchomień. Nie były one niestety związane z zużyciem zasobów typu RAM oraz CPU. Do klasyfikacji został użyty algorytm drzew decyzyjnych. Uzyskał on wysoką (ok. 97%) trafność co może być inspiracją przy wyborze algorytmów uczenia maszynowego.

Potencjalnie interesującą pracą jest [3], tutaj z kolei problemem był wybór parametrów dla przetwarzania przy użyciu technologii Spark. Autorzy chcieli wybrać odpowiednie ustawienie środowiska dla jak najlepszego zutylizowania zasobów korzystając z uczenia maszynowego. Spowodować by to miało przyspieszenie wykonywania się programów. Jednak można zauważyć, że uczenie maszynowe dotyczyło regresji, w celu przewidzenia czasu wykonywania się aplikacji przy zadanych parametrach na podstawie historycznych metryk. Ponieważ w pracy jest założone wykorzystanie istniejących implementacji algorytmów uczenia maszynowego w języku Python, to użyteczną informacją z tego artykułu jest fakt wykorzystania biblioteki Sklearn.

Nie udało się znaleźć prac, które używałby szeregów czasów (ang. *time series*) zużycia zasobów do klasyfikacji typu wykonywanej aplikacji w systemie. Przez to kierunkiem badań w pracy będzie analiza istniejących algorytmów mierzących podobieństwo między szeregami czasowymi oraz zostanie przeprowadzona klasyfikacja tych danych z wykorzystaniem uczenia maszynowego. W tym celu będą zbadane specyficzne dla tego problemu klasyfikatory i ich efektywność dla przypadku badanego w pracy, używając przetworzonego zbioru danych.

2.1 Podobieństwo szeregów czasowych - przegląd algorytmów

Na przestrzeni lat powstało bardzo wiele algorytmów zajmujących się porównywaniem szeregów czasowych. W znaczącej mierze różnią się one definicją podobieństwa. Przykładowo, jeśli są dwa szeregi i jeden jest taki sam jak drugi, tylko ma dwa razy większą amplitudę, to część podejść określiłoby je jako identyczne, a część jako zupełnie różne. Dlatego pierwszym krokiem do wybrania

odpowiedniego algorytmu jest ustalenie co znaczy podobieństwo w danym zbiorze danych i jakie będzie zastosowanie wybranego podobieństwa. Kolejne podrozdziały opisują kilka popularnych algorytmów obliczających podobieństwo.

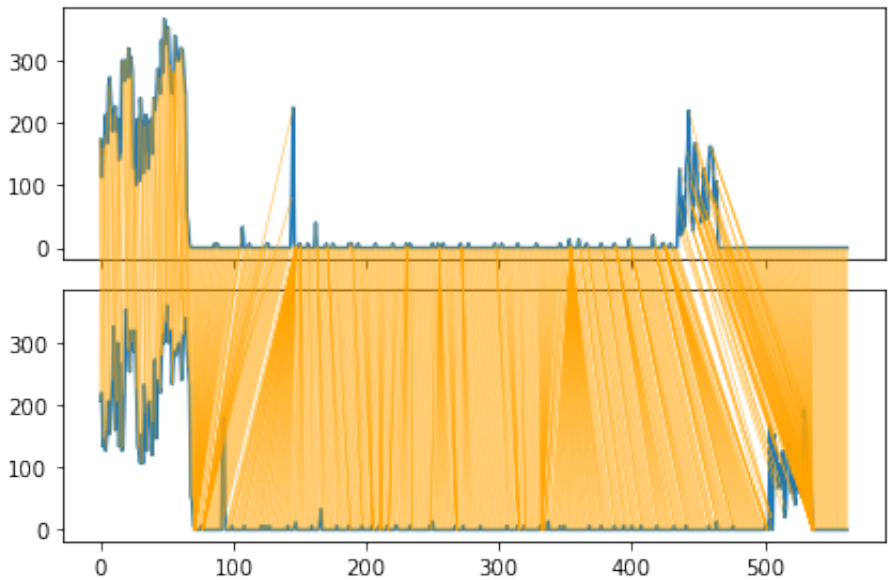
2.1.1 Odległość Euklidesowa

Odległość Euklidesowa jest najbardziej podstawową miarą podobieństwa. Jest to pierwiastek sumy długości odcinków między odpowiadającymi sobie punktami. Miara nie nadaje się do dużej części zastosowań związanych z obliczaniem podobieństwa szeregów czasowych, gdyż wymaga ona, aby szeregi były tej samej długości. Dodatkowo, jest bardzo podatna na wartości odstające, nie znajdzie również podobieństwa między dwoma, różnie przeskalowanymi wykresami. Miara ta jest jednak wykorzystywana w innych algorytmach takich jak Dynamic Time Warping.

2.1.2 Dynamiczna transformata czasowa (Dynamic Time Warping)

Dynamic Time Warping [6] jest jednym z najczęściej wykorzystywanych algorytmów liczących podobieństwo szeregów czasowych. Stosowano go często m.in do rozpoznawania mowy. W przeciwieństwie do odległości Euklidesowej można go wykorzystać przy szeregach o różnej długości. Algorytm zakłada znalezienie takiego dopasowania punktów jednego szeregu do drugiego, który będzie miał najmniejszy koszt i spełnia wszystkie warunki. Kosztem jest suma odległości między połączonymi punktami. Odległości te liczone są najczęściej za pomocą odległości Euklidesowej, jednak można wykorzystać każdą inną miarę liczącą dystans między dwoma punktami np. odległość Manhattan, czy współczynnik Gowera. Warunki dopasowania punktów są następujące:

- Pierwsze oraz ostatnie punkty przebiegów są dopasowane do siebie
- Dopasowania nie mogą się ze sobą krzyżować
- Podczas dopasowania nie można wykonać zbyt długich kroków



Rysunek 2.1: Przykład dopasowania punktów dla dwóch różniących się od siebie szeregów czasowych

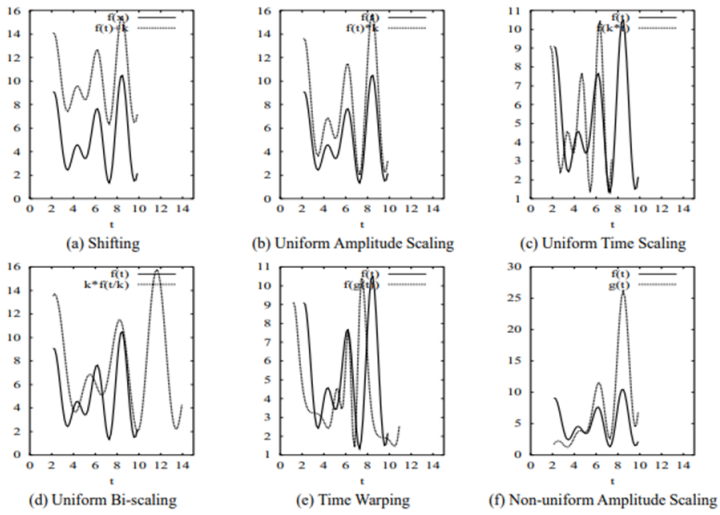
2.1.3 Najdłuższy wspólny podciąg (Longest Common Subsequence)

Kolejny algorytm [7] skupia się na znalezieniu części wspólnych szeregów, pomijając te które nie są podobne. Polega na znalezieniu najdłuższego wspólnego podciągu w dwóch szeregach czasowych. Podciąg określa się wspólnym podciągiem dwóch szeregów, jeżeli jest podciągiem zarówno pierwszego jak i drugiego szeregu. Elementy podciągów nie muszą znajdować się bezpośrednio obok siebie. Podobieństwo szeregów liczy się przez podzielenie dwukrotności długości wspólnego podciągu przez sumę długości obu szeregów. Część implementacji algorytmu podaje podobieństwo jako iloraz długości wspólnego podciągu i długości krótszego z dwóch szeregów.

2.1.4 Landmark Similarity

Landmark Similarity to algorytm obliczania podobieństwa, który został opisany w artykule[8]. Algorytm opiera się na założeniu, że ludzie uważają dwa wykresy za podobne, jeżeli ich charakterystyczne punkty są podobne, a reszta wykresu to krzywe łączące te punkty. Bierze on pod uwagę jedynie kształt wykresu.
Metoda zakłada zidentyfikowanie cech, które pozostają niezmienne po wykonaniu następujących transformacji

1. Przesunięcie (Shifting)
2. Jednolite Skalowanie Amplitudy (Uniform Amplitude Scaling)
3. Jednolite Skalowanie Czasu (Uniform Time Scaling)
4. Jednolite Bi-skalowanie (Uniform Bi-scaling)
5. Dopasowanie Czasu (Time Warping)
6. Niejednolite Skalowanie Amplitudy (Non-uniform amplitude scaling)



Rysunek 2.2: Transformacje w Landmark Model [8]

Przy wyborze odpowiednich cech charakterystycznych bierze się pod uwagę do czego ma służyć algorytm. Różne cechy są wykorzystywane do różnych zastosowań, od prostych (minima lokalne, maksima lokalne, punkty przełamania) do bardziej skomplikowanych. Dokładniejsze wyniki osiąga się używając większej liczby typów cech, jednak nie zawsze jest to potrzebne. Najczęściej minima

i maksima lokalne oraz punkty przełamania są wystarczające. Przed obliczeniem podobieństwa wygładza się wykresy, tak by mniej znaczące cechy nie były brane pod uwagę. Następnie porównuje się sekwencje cech ze sobą. Algorytm nie jest niestety powszechnie używany i nie udało się znaleźć jego implementacji w bibliotekach popularnych języków programowania.

2.2 Klasyfikacja szeregów czasowych

Klasyfikacja to jeden z znanych problemów, który jest rozwiązywany przy pomocy uczenia maszynowego. Jej celem jest przypisanie rekordów zbioru danych do odpowiedniej klasy. Klasyfikacja używa odpowiedniej funkcji mapującej zbiór cech na klasę, gdzie funkcja ta nauczona jest na zbiorze danych treningowych [9]. Przykładowe algorytmy, które rozwiązują ten problem to *Support Vector Machine (SVM)*, *Drzewa decyzyjne*, *KNN*. Chcąc ich użyć do danych, które są szeregami czasowymi, może wystąpić potrzeba ich lekkiej modyfikacji.

Szeregi czasowe od klasycznego zbioru danych rozróżnia ich struktura. Nie jest to zbiór cech, lecz pomiary zebrane w określonym oknie czasowym, gdzie ważne mogą być takie informacje jak amplituda czy kształt jaki tworzy wykres na podstawie tych danych. Dodatkowo, klasa może być opisywana wieloma różnymi szeregami. Przykładowo, badanym problemem może być określenie pory roku na podstawie dziennych pomiarów temperatury i wielkości opadów. Jednym wymiarem wtedy jest średnia temperatura i jej zmiana w ciągu miesiąca, a drugim wymiarem średnia intensywność opadów. Kolejnym wyzwaniem w przypadku klasyfikacji szeregów czasowych jest sytuacja gdzie badane zjawisko było mierzone przez nierówny czas. Wcześniej wspomniana pogoda ma określone okno pomiarowe (np. 24 godziny), ale nie zawsze tak musi być. Jeżeli badany jest stan systemu podczas działania aplikacji, to długość obserwacji zależy od czasu działania tego programu. Algorytm badający szeregi czasowe musi być odpowiednio przystosowany, aby poprawnie interpretować tego typu dane. Biorąc to pod uwagę, została podjęta próba znalezienia odpowiednio przystosowanych algorytmów do klasyfikacji szeregów czasowych.

2.2.1 Oceny jakości klasyfikatorów

W przypadku klasyfikacji, ocenę jakości modelu klasyfikacyjnego wykonuje się w oparciu o dane testowe. Znając etykiety przypisane do próbek, można zastosować przygotowany klasyfikator i porównać przypisane klasy z rzeczywistymi. W ten sposób zlicza się liczbę poprawnych i niepoprawnych klasyfikacji. Jedną z najczęściej wykorzystywanych form przedstawienia wyniku jest tak zwana macierz pomyłek. Jest to macierz kwadratowa $m \times m$ gdzie m oznacza liczbę klas. Przykładowo, tablica 2.1 przedstawia tą macierz dla problemu binarnego 2.1. W literaturze elementy macierzy często określa się przy pomocy następujących symboli, gdzie P to klasa pozytywna, a N to negatywna [10]:

- TP*(ang. true positive)* - liczba poprawnie zaklasyfikowanych rekordów testowych do klasy P
- FN*(ang. false negative)* - liczba błędnie zaklasyfikowanych rekordów testowych klasy P do klasy N
- FP*(ang. false positive)* - liczba błędnie zaklasyfikowanych rekordów testowych z klasy N do klasy P
- TN*(ang. true negative)* - liczba poprawnie zaklasyfikowanych rekordów testowych do klasy N

Tablica 2.1: Macierz pomyłek dla problemu klasyfikacji binarnej

Klasa rzeczywista	Klasa przewidywana		Suma
	P	F	
P	TP (true positives)	FP (false positives)	TP+FP
F	FN (false negatives)	TN (true negatives)	FN+TN

Struktura ta pozwala na analizę jakości klasyfikatorów. Zależnie od problemu można być zainteresowanym innymi błędami. Przykładowo, w medycynie określając czy badany pacjent jest chory, bardziej użyteczny jest algorytm, który wykrywa chorobę u pacjentów zdrowych (błąd FP), niż taki który nie wykryje choroby u faktycznie chorej osoby (Błąd FN). Jeżeli celem jest ogólne sprawdzenie jakości klasyfikacji dobrą miarą jest **accuracy** (po polsku dokładność/trafność) i można ją policzyć z następującego wzoru:

$$accuracy = \frac{liczba\ poprawnie\ zaklasyfikowanych\ rekordw\ testowych\ (TP)}{łączna\ liczba\ rekordw\ testowych} \tag{2.1}$$

Poza dokładnością często używane są jeszcze inne miary i zależnie od potrzeb można je wykorzystać dla badanego problemu. Należą do nich między innymi sensivity (po polsku czułość), FP rate (po polsku współczynnik FP), specifity (po polsku specyficzność), precision (po polsku precyzja), recall (po polsku recall) oraz F-measure (po polsku F-miara)[10].

$$sensivity = \frac{TP}{P} \tag{2.2}$$

$$FP\ rate = \frac{FP}{N} \tag{2.3}$$

$$specifity = \frac{TN}{N} \tag{2.4}$$

$$precision = \frac{TP}{TP\ +\ FP} \tag{2.5}$$

$$recall = \frac{TP}{TP\ +\ FN} \tag{2.6}$$

$$F - measure = \frac{2\ *\ precision\ *\ recall}{precision\ *\ recall} \tag{2.7}$$

W pracy będą stosowane miary accuracy oraz F-measure. do oceny algorytmów, które zostaną wykorzystane do klasyfikacji danych w postaci szeregów czasowych.

2.2.2 ROCKET

ROCKET, czyli **R**and**O**m **C**onvolutional **KE**rnel **T**ransform wykorzystuje, jak nazwa wskazuje, losowe kernele konwolucyjne do transformacji szeregów czasowych, w celu użycia wyniku tej operacji jako cech w klasyfikatorze [11]. Autorzy wspierali się tym, że zastosowanie neuronowych sieci konwolucyjnych jest efektywne dla szeregów czasowych. Również podali oni argumenty za tym, że wykorzystanie losowych kerneli konwolucyjnych oraz użycie ich wyjściowych transformacji jako cech wejściowych dla innego klasyfikatora jest wykorzystywane w przypadku analizy obrazów.

Metoda ta wykorzystuje ekstremalną liczbę kerneli (domyślnie używali 10.000), w których każdy generuje 2 wartości, w ten sposób każdy szereg czasowy ma wytworzoną dużą liczbę cech (w przypadku domyślnych ustawień 20.000 cech). Autorzy twierdzą, że im więcej jąder konwolucyjnych tym większa jakość klasyfikacji, kosztem czasu przetwarzania. Algorytm klasyfikacji użyty po

transformacjach może być dowolny, jednak twórcy polecają klasyfikatory liniowe (domyślnie użyta została *regresja grzbietowa* (ang. Ridge Regression)).

Przy generowaniu kerneli w celu zachowania jak największej losowości użyto 5. parametrów z następującymi zasadami ich wyboru:

- Długość kernela - jest to długość wektora wag, która jest wybierana ze zbioru {7, 9 ,11}.
- Wartości wag - losowe wartości generowane z rozkładu normalnego, następnie centrowane wokół średniej otrzymanego zbioru wartości.
- Bias - czyli liczba dodana do wyniku transformacji pojedynczej wartości. Służy to urozmaicenia potencjalnie podobnych kerneli, przesuwając wartości ponad lub poniżej 0. Losowana z rozkładu jednorodnego $b \sim U(-1, 1)$
- Dilation - ten parametr "rozszerza" kernel. Przykładowo, jeżeli ten parametr jest równy 2, to element wejściowy będzie poddany transformacji z co drugim elementem, zamiast z bezpośrednimi sąsiadami. Umożliwia to wykrywanie wzorów na różnych częstotliwościach szeregu czasowego. Wybierany jest on w skali wykładniczej $d = 2^x$, $x \sim U(0, A)$, $gdzieA = \log_2 \frac{l_{input}-1}{l_{kernel}-1}$. Wzór ten zapewnia zgodność tego parametru z długością szeregu czasowego.
- Padding - po wygenerowaniu kernela podejmowana jest decyzja z równym prawdopodobieństwem, czy zostanie zastosowany padding lub nie. Jego działanie dodaje na początek i koniec szeregu 0 w celu ustawienia środka kernela na pierwszym faktycznym elemencie szeregu.

Autorzy przeprowadzili eksperymenty, które doprowadziły do powyższych kryteriów wyboru parametrów. Zgodnie z wynikami dają one najlepsze rezultaty [11].

Główną zaletą tego podejścia poza trafnością, która jest na równi (jak nie lepsza) z innymi nowoczesnymi metodami klasyfikacji szeregów czasowych, jest czas przetwarzania. Złożoność tego algorytmu zależy głównie od liczby kerneli, a poza tym od liczby przykładów oraz długości przebiegów czasowych. Drugim czynnikiem wpływającym na czas potrzebny na wykonanie obliczeń jest złożoność użytego klasyfikatora, więc w przypadku regresji grzbietowej zależy ona od wielkości zbioru i liczby badanych cech. Poprzez eksperymenty w pracy, autorzy pokazali, że w porównaniu z innymi algorytmami czas trwania klasyfikacji jest krótszy od reszty dla tej samej liczby przykładów z utrzymaniem podobnej trafności. Takie zachowanie może być dobrą cechą dla przyszłych eksperymentów wykonanych w pracy.

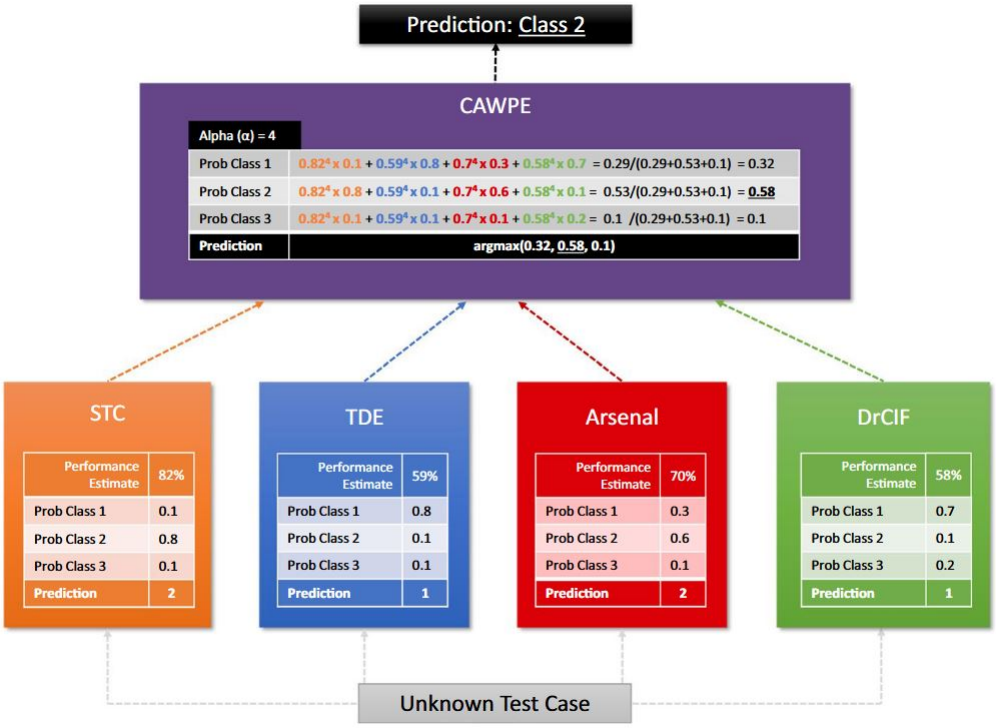
Podsumowując tą sekcje, metoda ROCKET zdaję się być dobrym algorytmem do sprawdzenia pod kątem klasyfikacji szeregów czasowych. Autorzy artykułu stwierdzili, że implementacja w ramach której wykonywali eksperymenty była wykonana w języku Python, więc powinna być możliwość wykorzystania jej w badaniach. Jedyne czego brakowało w artykule to wykorzystanie algorytmu dla wielowymiarowych szeregów czasowych.

2.2.3 HIVECOTE2

HIVECOTE, czyli Hierarchical Vote Collective of Transformation-based Ensembles to heterogeniczny meta zbiór metod klasyfikacji szeregów czasowych [12]. Każdy z zawartych algorytmów dotyczy innej domeny. Można w tej metodzie znaleźć takie algorytmy jak *shapelets*, słowniki bazujące na modelu "bag-of-words" oraz metody interwałowe. Zgodnie z badaniami autorów pracy, HIVECOTE2 średnio jest jednym z najlepszych klasyfikatorów szeregów czasowych. Konkretne modele zawarte w ramach drugiej wersji tego algorytmu to:

- *Shapelet Transform Classifier*
- Dostosowana wersja wcześniej wspomnianego algorytmu *ROCKET* nazwana *ARSENAL*
- *Temporal Dictionary Ensemble (TDE)*
- *Diverse Representation Canonical Interval Forest (DrCIF)*

Każdy z tych algorytmów jest uczony osobno i dodatkowo, na potrzeby ostatecznego wyniki musi zwrócić estymacje swojej trafności na nieznanych im wcześniej danych. Podczas predykcji, każda z metod zwraca prawdopodobieństwo przynależności badanego szeregu do danej klasy po czym obliczana jest średnia ważona. Wagi w średniej to wcześniej obliczona dokładność poszczególnych algorytmów, która jest zmniejszona poprzez potęgowanie przez zadaną wartość (domyślnie jest to 4). Poniższy rysunek 2.3 pokazuje przykładowe działanie tego algorytmu w praktyce, dla problemu trzy-klasowego. Każdy z nauczonych algorytmów dostaje na wejście nieznany wcześniej przypadek do zaklasyfikowania. Określają one prawdopodobieństwo przynależności do wszystkich trzech klas i wraz z informacją o ich trafności przekazują je na wyjście do następnego etapu. W tym momencie posiadając wszystkie dane algorytm liczy średnią ważoną, gdzie dokładność jest pomniejszana przez zadaną wartość parametru alfa i wykorzystana jako waga do każdej wartości procentowej. W ten sposób przypadek zostaje zaklasyfikowany do klasy, która osiągnęła najwyższą wartość średniej, gdzie na przykładzie jest to klasa 2.



Rysunek 2.3: Przykładowe działanie algorytmu HIVECOTE2 [12]

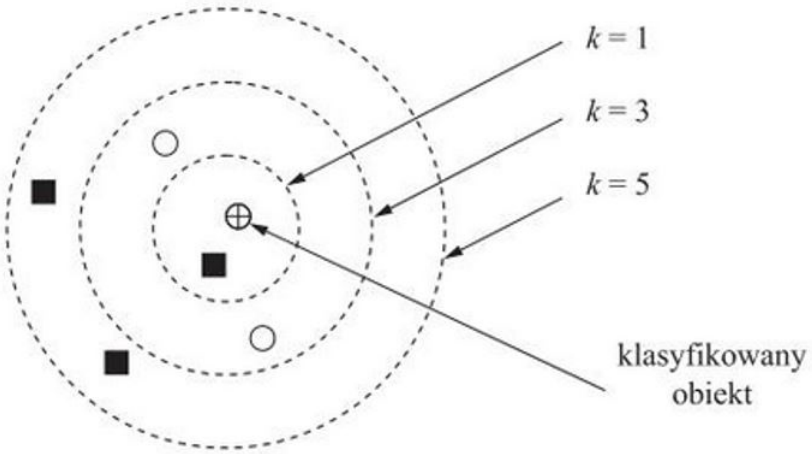
Jednym z zarzutów dla metody HIVECOTE2 jest jego długi czas uczenia. W porównaniu z algorytmem ROCKET można zauważyć, że algorytm HIVECOTE2 jest wolniejszy. W pracy autorzy zmieścili porównanie, gdzie ROCKET wykonuje się ponad sto razy szybciej od algorytmu HIVECOTE2. Jednak twórcy umożliwiają ograniczenie czasu procesowania co może pozwolić nam na wygodne wykorzystanie tej metody do badań, kosztem dokładności obliczeń.

Podsumowując, ten algorytm jest warty uwagi i należy zwrócić uwagę na to, że był testowany również na danych wielowymiarowych, co jest zgodne z danymi wykorzystywanymi podczas badań.

2.2.4 KNN-DTW

Poza wyspecjalizowanymi podejściami do klasyfikacji szeregów czasowych, warto sprawdzić czy bardziej ogólne metody nadają się do badania tego problemu. Jednym z takich algorytmów jest KNN albo K-najbliższych sąsiadów (ang. K-nearest neighbor). Jest on oparty na analizie przypadku (ang. instance-based classification methods), czyli zamiast konstruować model klasyfikacyjny działa on wtedy, kiedy zachodzi potrzeba zaklasyfikowania nowego przypadku [10].

Ta metoda przechowuje wszystkie dostępne rekordy i nadaje klasę nowemu przypadkowi na podstawie zmierzonego podobieństwa. K jest to liczba sąsiadów, których "głos" jest wzięty pod uwagę podczas predykcji. Przykładowo, jeżeli k = 3 to koło dookoła nowego rekordu, gdzie promieniem jest podobieństwo, będzie zawierało w sobie trzy inne przykłady. Te przykłady są wzięte do głosowania aby określić przynależność do klasy badanego przypadku [9].



Rysunek 2.4: Wpływ wartość parametru k na klasyfikacje [10]

Zasadniczym problem w tej metodzie jest wybór odpowiedniej liczby sąsiadów oraz miary podobieństwa określającej kto jest "najbliższym sąsiadem" badanego przykładu. Na rysunku 2.4 widać wpływ parametru k na klasyfikacje. W przypadku k = 1, obiekt zostałby zaklasyfikowany jako "kwadrat". Jednak dla k=3, większość sąsiadów to "okręgi", więc ta klasa by została wybrana. Można zauważyć, że im mniejsza liczba sąsiadów, tym bardziej podobne obiekty są brane pod uwagę. Sprawia to, że ten parametr należy wybrać eksperymentalnie, zgodnie z intuicją badacza. Używając tego algorytmu do przebiegów czasowych, konieczny jest wybór odpowiedniej miary podobieństwa, pasującej do tego problemu. W sekcji 2.1 zbadano znane podejścia i najbardziej wyróżniającą się miarą jest *Dynamic Time Warping*. Użycie tego podejścia w połączeniu z algorytmem K-NN jest sprawdzoną metodą i udowodniono, że jakość klasyfikacji w tym przypadku jest bardzo dobra, osiągająca ok. 80% trafności na badanych danych [13]. Autorzy pracy również wzięli pod uwagę inne miary podobieństwa, jednak w przypadku klasyfikacji, wykorzystanie DTW jako algorytm podobieństwa, osiągnął najlepsze wyniki. Można znaleźć implementacje algorytmu KNN w języku Python co daje nam możliwość użycia go w eksperymentach.

2.3 Metody normalizacji danych

W pracach zajmujących się uczeniem maszynowym bardzo często występuje normalizacja danych. Przykładowo, [2] wykorzystuje normalizacje Z-score, a w artykule [3] twórcy użyli skalowania Min-Max. Wykonuję się ją na etapie przygotowywania danych (ang. Preprocessing) najczęściej w celu poprawienia wyników algorytmów. Zdarza się, że algorytmy uczenia maszynowego wymagają znormalizowanych danych dla poprawnego działania [14]. Dzięki temu procesowi można otrzymać wartości, które w zbiorze danych będą miały podobną skale. W tej sekcji opisane zostaną popularne metody normalizacji danych, czyli **Min-Max** oraz **Z-score**. Jedna z tych metod zostanie użyta na danych wykorzystywanych w tej pracy.

2.3.1 Min-Max

Technika normalizacji Min-Max polega na odjęciu od danej liczby A_i, najmniejszej wartości z dziedziny A, a następnie podzielenie wyniku przez największą wartość z tej samej dziedziny pomniejszoną przez minimum. Tą metodę można przedstawić za pomocą następującego wzoru:

$$A'_i = \frac{A_i - min(A)}{max(A) - min(A)} \tag{2.8}$$

W ten sposób uzyskujemy przeskalowane wartości z zakresu od 0 do 1. Jest to dobra technika, jeżeli nie znamy rozkładu wartości naszych danych lub jeżeli wiemy, że nie jest to rozkład normalny.

2.3.2 Z-score

Ten sposób normalizacji to skalowanie poprzez odjęcie od danej liczby A_i, średniej wszystkich wartości z dziedziny A, a następnie podzielenia wyniku przez odchylenie standardowe zbioru. Wzór prezentuje się następująco:

$$A'_i = \frac{A_i - mean(A)}{std(A)} \tag{2.9}$$

W wyniku działania tej techniki, normalizowany zbiór nie zostanie ograniczony przez predefiniowany zakres wartości. W efekcie tej transformacji otrzymujemy zbiór wartości, którego średnia jest równa 0, a odchylenie standardowe 1. Użycie tej metody jest preferowane, jeżeli znany jest rozkład wartości i jest to rozkład normalny.

Rozdział 3

Opis danych

3.1 Struktura i schemat danych surowych

Przed przeprowadzeniem analizy i eksperymentów potrzebne jest zdobycie danych wejściowych, odpowiednich dla badanego problemu. W pracy pilotażowej [5] został utworzony taki zbiór danych i został udostępniony w internecie. Znaleźć je można na repozytorium w serwisie GitHub.com utworzonym przez autorów poprzedniej pracy[1] i zostaną one wykorzystane w tej pracy. Dane te zostały zebrane podczas działania aplikacji w środowisku Apache Spark z każdego skonfigurowanego węzła i zapisane w formacie CSV w odpowiednim folderze. W sumie dostępnych jest 33600 surowych plików, gdzie każdy plik to osobny przebieg wykonywania się funkcji. W zbiorze danych możemy wyróżnić pięć typów przetwarzania w ramach których zaimplementowane są konkretne UDF'y:

- aggregation
- filtering
- filtering-aggregation
- filtering-aggregation-join
- filtering-join

Struktura folderów w których znajdują się pliki z danymi prezentuje się następująco: "<typ funkcji>/<UDF>/<konfiguracja klastra>/source-data/node-<id węzła>/<data uruchomienia>.csv". Przykładowo, dla paru pierwszych plików poniżej znajduje się drzewo folderów:

[1] `https://github.com/pientaa/spark-udf-characteristics`

```
├──aggregation
│   ├──avgNetProfitGroupedBySoldDate
│   │   ├──CPU-data-vol-1GB-5nodes
│   │   │   └──source-data
│   │   │       ├──node-11
│   │   │       │       07_08_2021_23_43_34.csv
│   │   │       │       ...
│   │   │       │       08_08_2021_00_23_30.csv
│   │   │       ├──node-12
│   │   │       │       07_08_2021_23_46_44.csv
│   │   │       │       ...
│   │   │       │       08_08_2021_00_26_40.csv
│   │   │       ├──node-14
│   │   │       │       07_08_2021_23_45_13.csv
│   │   │       │       ...
│   │   │       │       08_08_2021_00_25_09.csv
│   │   │       │
│   │   │       ├──node-15
│   │   │       │       07_08_2021_23_55_41.csv
│   │   │       │       ...
│   │   │       │       08_08_2021_00_35_37.csv
│   │   │       │
│   │   │       └──node-20
│   │   │               07_08_2021_23_43_43.csv
│   │   │               ...
│   │   │               08_08_2021_00_23_38.csv
```

Rysunek 3.1: Struktura surowych danych

Każda funkcja została uruchomiona 25 razy w ramach zdefiniowanych konfiguracji klastra. Węzły, które są częścią klastra, raportowały informacje o zużyciu procesora i pamięci RAM w trakcie wykonywania się UDF'a dla utworzonych procesów. Wszystko to jest zebrane w plikach CSV o następującym schemacie:

- timestamp - znacznik czasowy w formacie "rok-miesiąc-dzień godzina:minuta:sekunda.milisekunda"
- PID - identyfikator procesu
- CPU - procent zużycia procesora przez proces, dla zadań wielowątkowych może przekroczyć 100%
- RAM - procent zużycia pamięci przez proces

Tablica 3.1 pokazuje zawartość jednego z surowych plików. Można w niej zauważyć powtarzające się wartości dla pola "timestamp" jak i powtarzające się identyfikatory procesów "PID". Widać również, że odstępy między znacznikami czasowymi, które wynoszą ok. $\frac{1}{15}$ sekundy. W celu zbadania charakterystyki danych wejściowych należy odpowiednio zgrupować dane i zsumować obserwacje, w celu uzyskania pojedynczych wartości dla każdego znacznika czasowego.

Tablica 3.1: Początek pliku aggregation/avgNetProfitGroupedBySoldDate/CPU-data-vol-1GB-5nodes/source-data/node-11/07_08_2021_23_43_34.csv

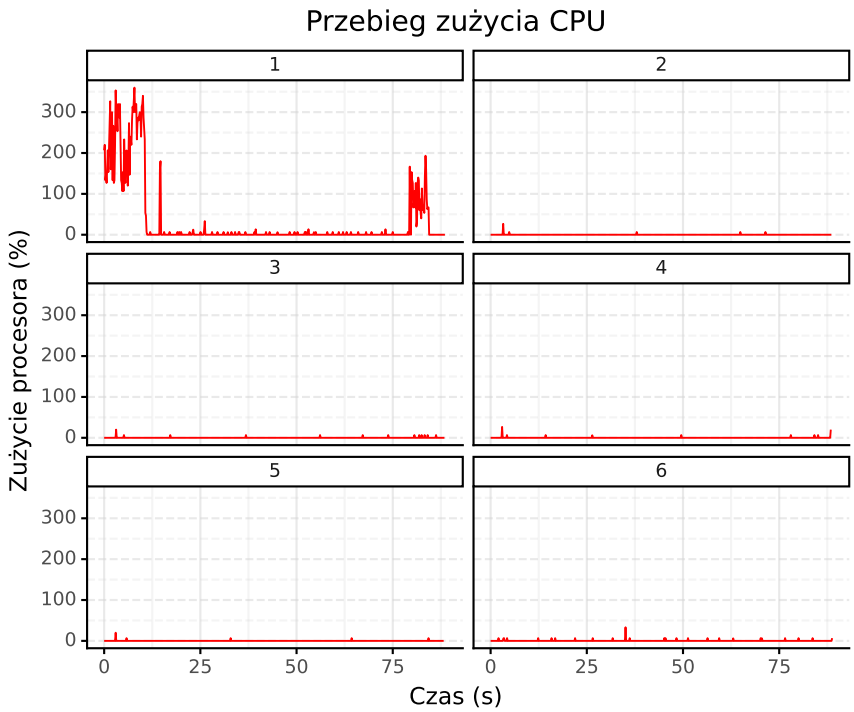
timestamp	PID	CPU	RAM
2021-08-07 23:43:34.842594	2219209	206.7	0.7
2021-08-07 23:43:34.842594	2125032	0.0	0.0
2021-08-07 23:43:34.842594	2125094	0.0	4.1
2021-08-07 23:43:34.998864	2219209	220.0	0.9
2021-08-07 23:43:34.998864	2125032	0.0	0.0
2021-08-07 23:43:34.998864	2125094	0.0	4.1
2021-08-07 23:43:35.156604	2219209	133.3	1.1
2021-08-07 23:43:35.156604	2125032	0.0	0.0
2021-08-07 23:43:35.156604	2125094	0.0	4.1

3.2 Charakterystyka danych wejściowych

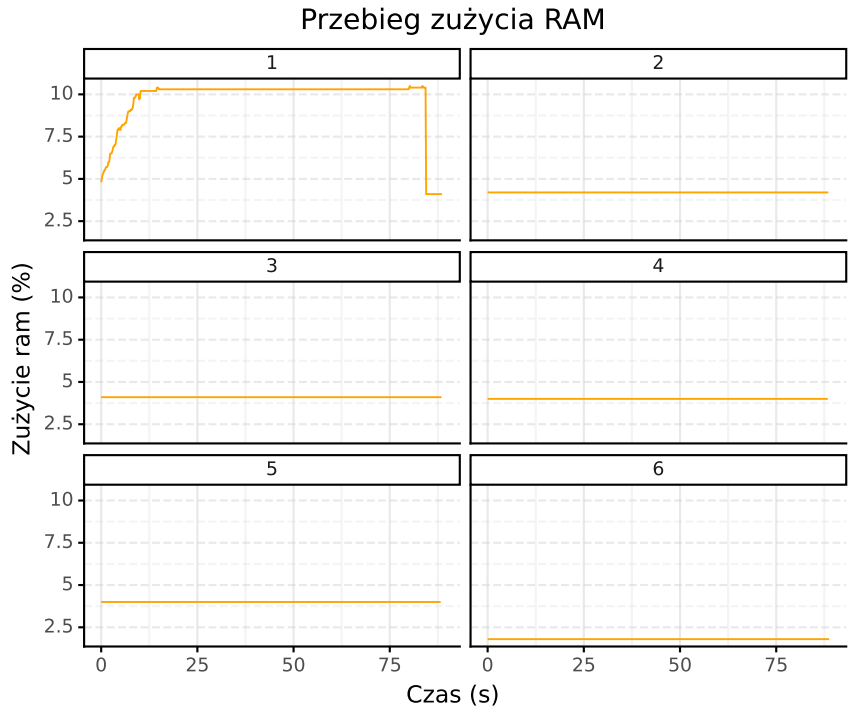
Dla każdej zdefiniowanej funkcji przygotowano sześć konfiguracji klastra. Na każdym konfiguracji wykonano dwadzieścia pięć eksperymentów. W konfiguracji ustawiano dwie zmienne - wielkość przetwarzanych danych oraz liczbę węzłów. Skutkiem takich ustawień było następujące sześć grup eksperymentów:

- Rozmiar danych: 1GB, 5 Węzłów + 1 Węzeł Master
- Rozmiar danych: 1GB, 7 Węzłów + 1 Węzeł Master
- Rozmiar danych: 1GB, 9 Węzłów + 1 Węzeł Master
- Rozmiar danych: 2GB, 5 Węzłów + 1 Węzeł Master
- Rozmiar danych: 2GB, 7 Węzłów + 1 Węzeł Master
- Rozmiar danych: 2GB, 9 Węzłów + 1 Węzeł Master

Węzeł Master decydował o przydzieleniu odpowiedniej liczby węzłów do pracy. W wyniku analizy danych z eksperymentów, okazało się, że wszystkie eksperymenty zawierały jeden wyróżniający się węzeł, który zużywał znacząco więcej CPU i RAM niż pozostałe. Z tego wynika, że tylko on otrzymał zadanie do wykonania. Na wykresach 3.2 i 3.3 można zauważyć, że pierwszy węzeł wykonywał pracę zużywając wielokrotnie więcej CPU i RAM niż pozostałe węzły. Węzeł szósty pełnił rolę mastera - komunikował się z węzłami pracującymi, nie wykorzystując przy tym w ogóle pamięci RAM. Pozostałe węzły informowały cyklicznie, że są w gotowości,



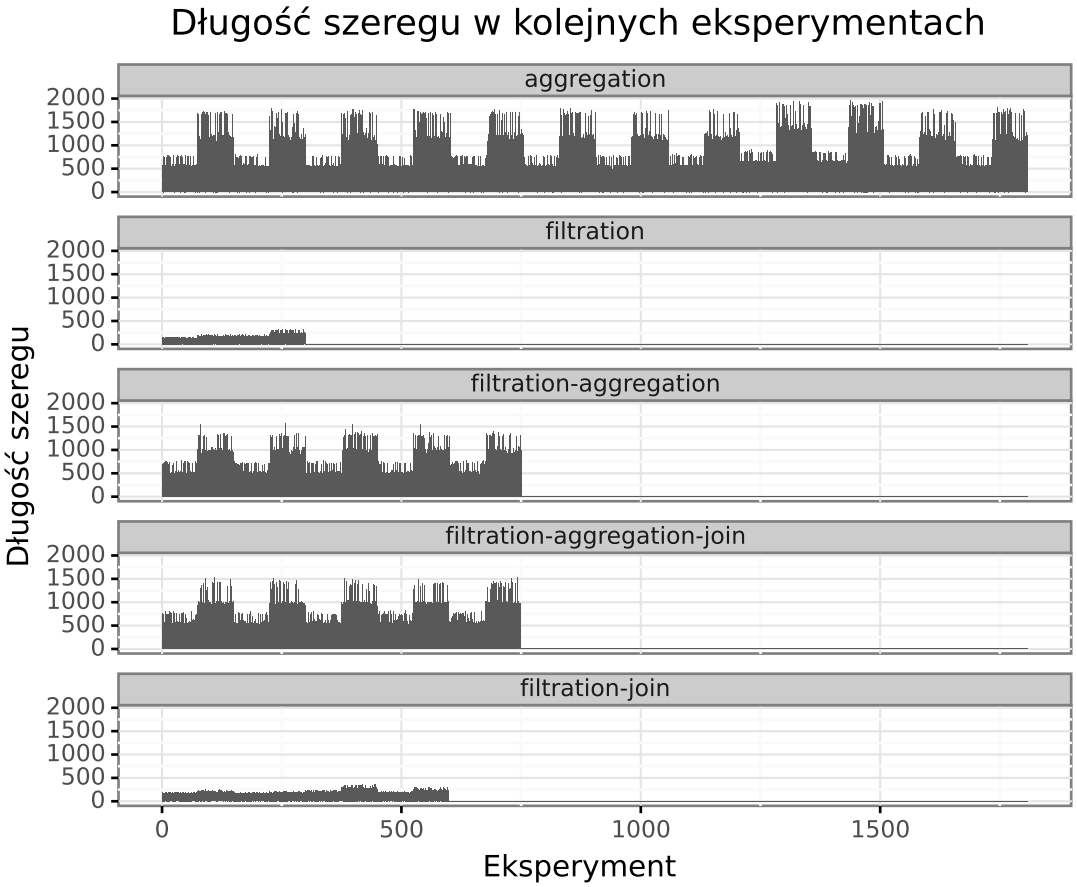
Rysunek 3.2: Zużycie CPU dla pierwszego eksperymentu z pięcioma węzłami pracującymi oraz jednym masterem



Rysunek 3.3: Zużycie RAM dla pierwszego eksperymentu z pięcioma węzłami pracującymi oraz jednym masterem

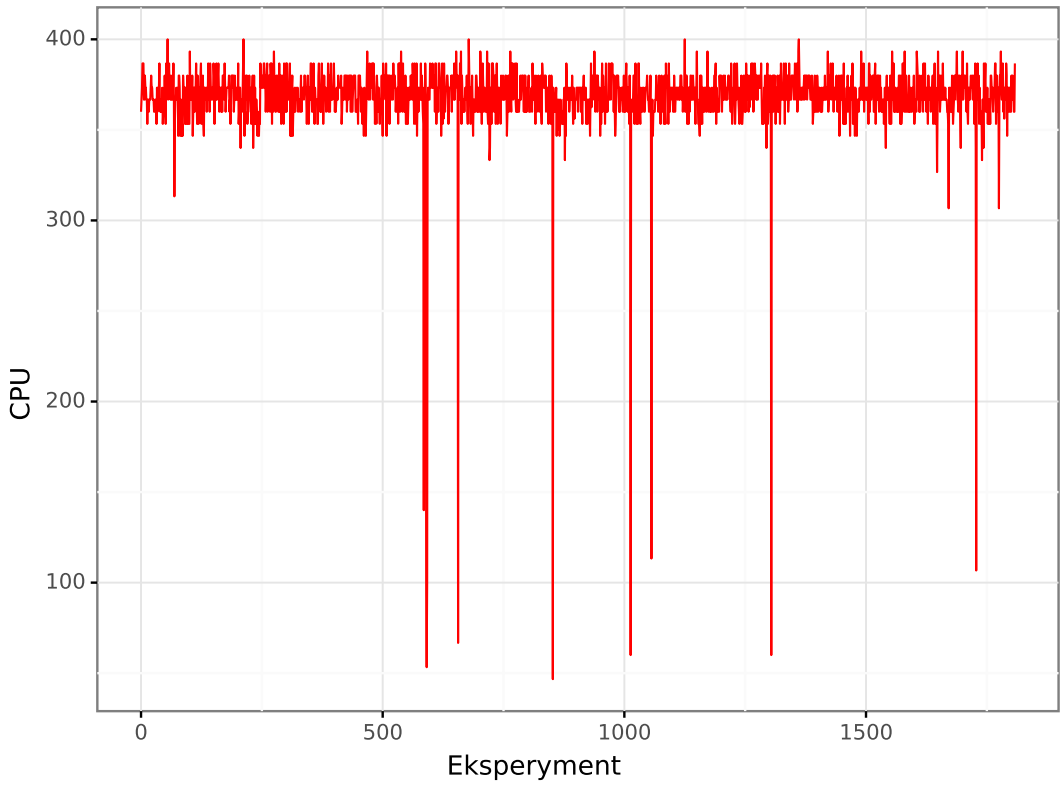
Liczba węzłów raczej nie wpłynie znacząco na podobieństwo szeregów, gdyż można zauważyć aktywność tylko jednego z węzłów pracujących. Reszta zostanie usunięta, czyli węzły pełniące rolę mastera oraz węzły nieaktywne nie będą brane do analizy. Rozmiar przetwarzanych danych wpływa jednak istotnie na długość wykonywania się funkcji, a co za tym idzie na część wykorzystywanych

algorytmów. Dodatkowo, nawet biorąc pod uwagę identyczną konfigurację dla dwóch kolejnych eksperymentów dotyczącą tej samej funkcji, długość szeregów czasowych okazała się zmienna. Analiza kolejnych typów funkcji wykazała, że długość szeregu jest jedną z cech różniących je od siebie. Szczególnie odróżnia ona filtrację oraz filtrację-join, które wykonują się znacznie szybciej niż pozostałe typy funkcji. Jest to widoczne na rysunku 3.4 przedstawiającym długość szeregu czasowego w kolejnych eksperymentach.



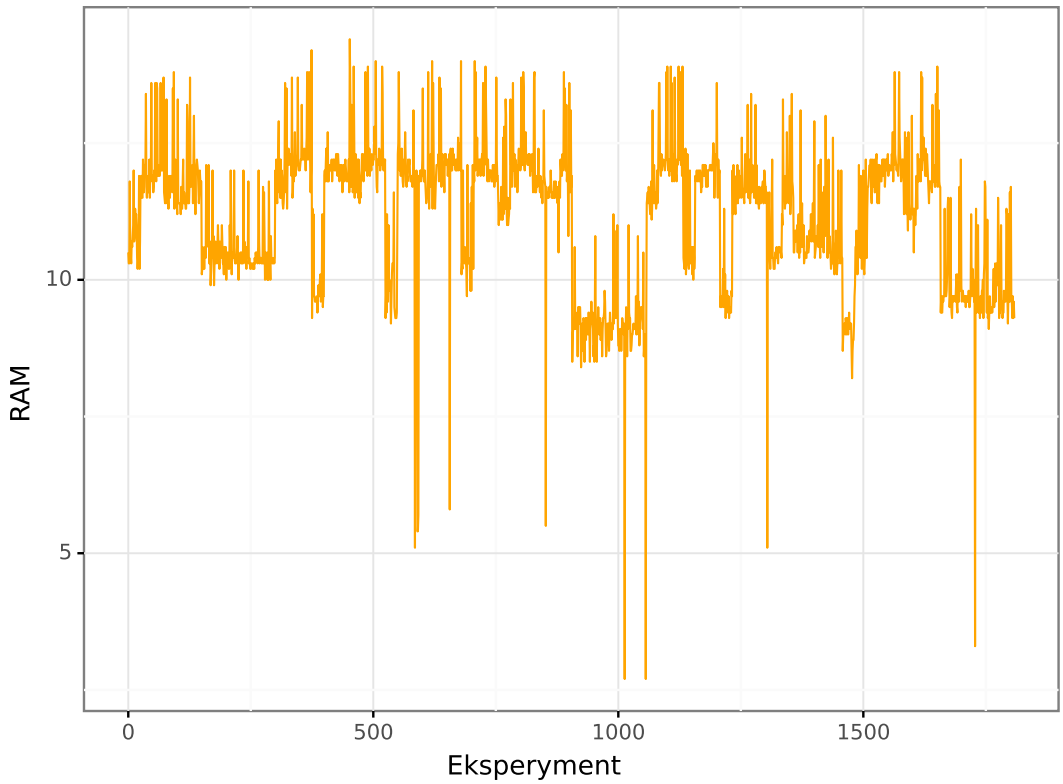
Rysunek 3.4: Długość szeregu/eksperyment

Dla dokładniejszej oceny danych przeanalizowano również maksymalne wartości RAM i CPU kolejnych eksperymentów dla danego typu funkcji. Na przykładzie agregacji można zauważyć, że dla CPU wartości nie były jednakowe, za to wahały się między 340 a 390. Było jednak kilka wartości odstających o zdecydowanie mniejszej wartości

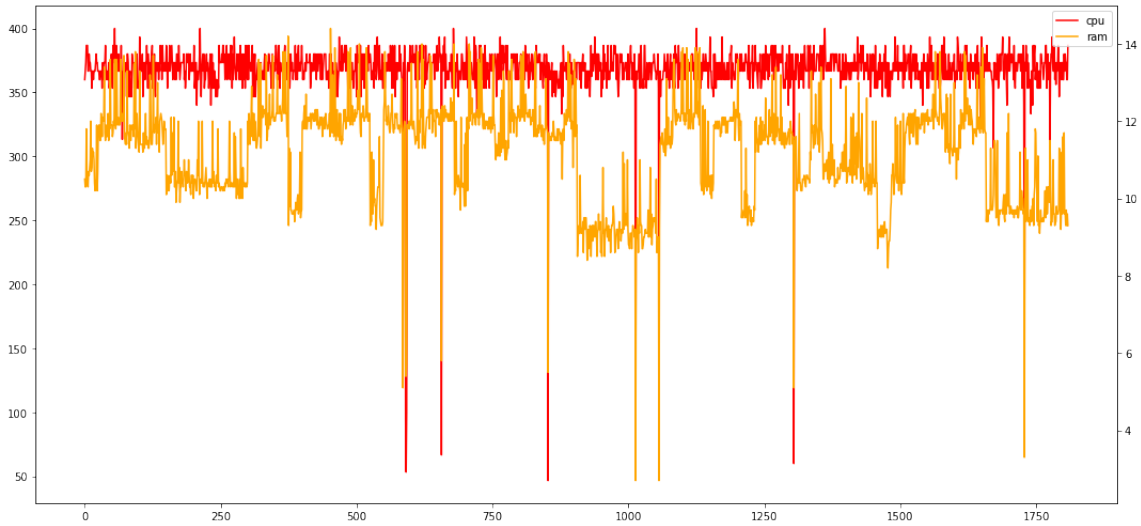


Rysunek 3.5: Agregacja - maksymalne CPU/eksperyment

Po analizie podobnego wykresu, tylko dotyczącego RAM, okazało się że wartości odstające pojawiły się w dokładnie tych samych eksperymentach.

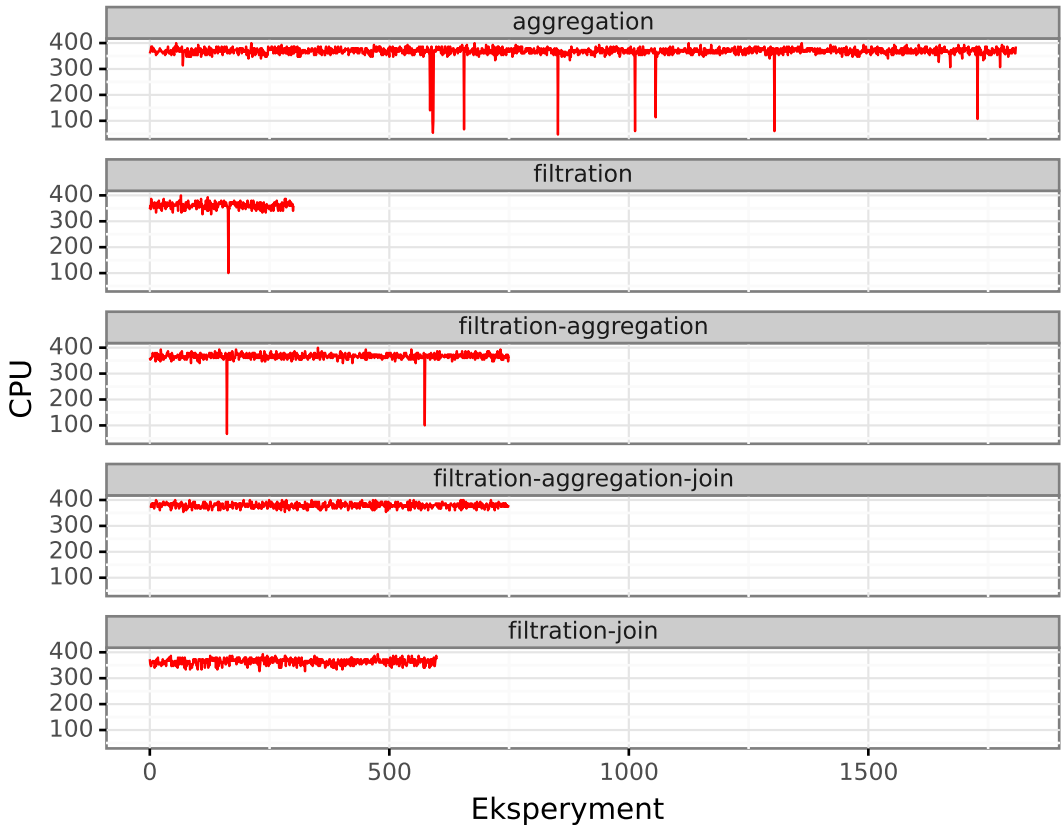


Rysunek 3.6: Agregacja - maksymalny RAM/eksperyment

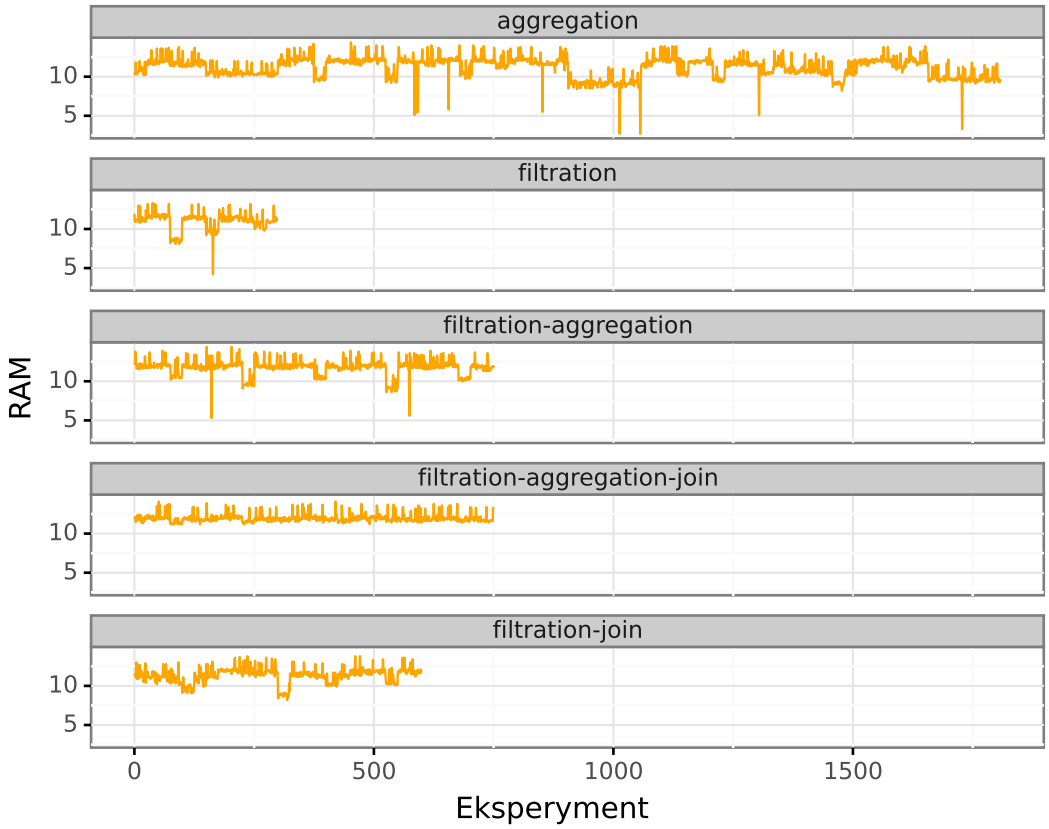


Rysunek 3.7: Agregacja - maksymalny RAM oraz CPU/eksperyment

Pozostałe typy funkcji miały podobne maksymalne wartości. Miały też w większości przynajmniej jedną wartość odstającą.

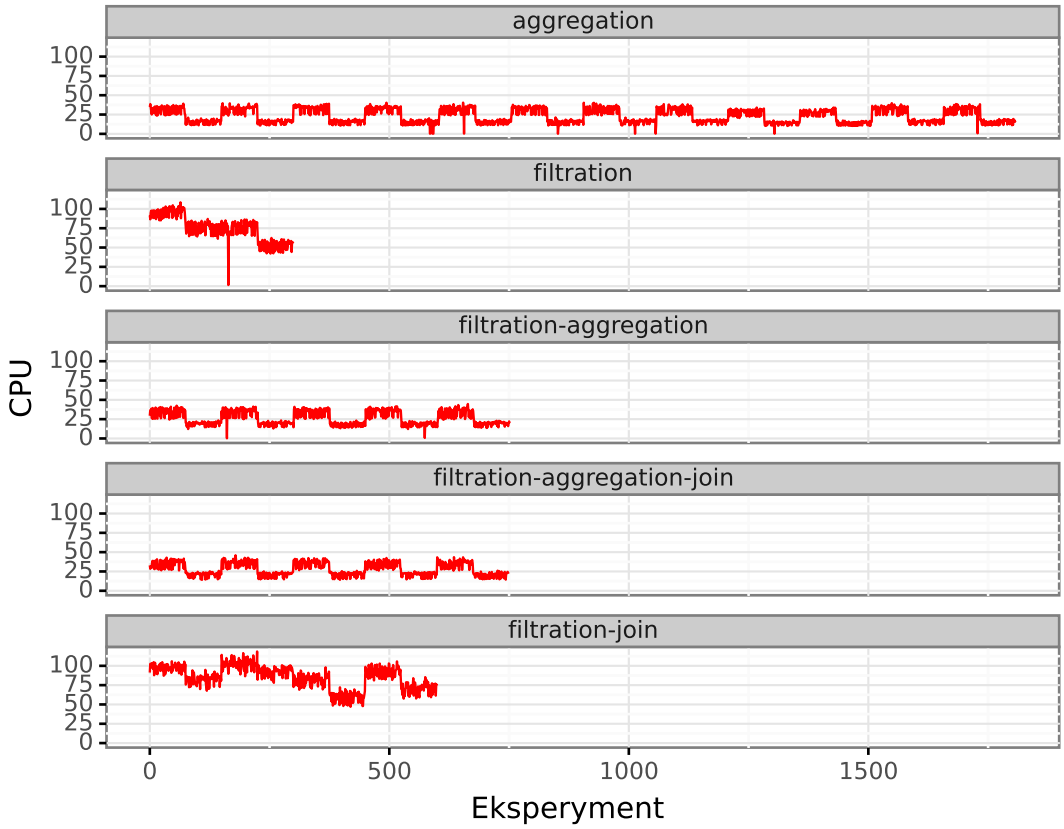


Rysunek 3.8: Maksymalne CPU/eksperyment

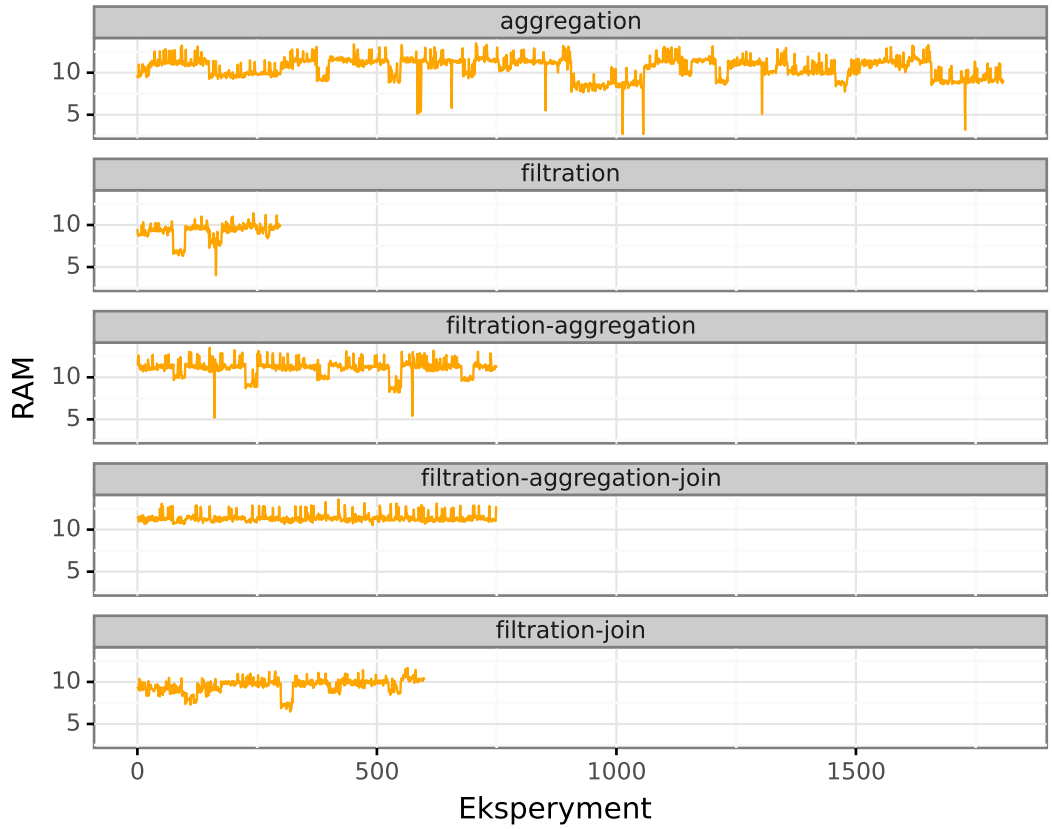


Rysunek 3.9: Maksymalny RAM/eksperyment

Przeanalizowane zostały również wartości średnie CPU i RAM. Największe różnice między funkcjami zaobserwowano dla wartości średniej CPU. Na rysunku 3.10 widać, że funkcje zawierające agregacje zużywają średnio mniej CPU niż filtracja i filtracja-join. Dodatkowo, warto zaobserwować, że jeśli chodzi o agregacje, filtracje-agregacje oraz filtracje-agregacje-join wartości średnie są bardzo regularne. Spadki i wzrosty wynikają głównie z tego, że każda funkcja występuje w wersji 1GB i 2GB. Nie dzieje się tak jednak z filtracją. Filtracja jest przedstawiona w zbiorze danych za pomocą jedynie dwóch UDF: filterCatalogSalesWhereProfitNegative i filterStoreSalesWhereProfitNegative. Wyraźnie widać, że jedna z nich używa więcej CPU od drugiej. Może to powodować trudności w identyfikacji filtracji w dalszych rozdziałach pracy.



Rysunek 3.10: Średnia CPU/eksperyment



Rysunek 3.11: Średnia RAM/eksperyment

3.3 Charakterystyka danych w kontekście wyboru algorytmu

Podsumowując, analiza danych wykazała, że porównując funkcje między sobą należy spodziewać się pojedynczych wartości odstających. Zostało również odkryte, że jedną z ważniejszych różnic między typami funkcji jest czas wykonywania, a co za tym idzie długość szeregu czasowego. Dodatkowo, tak jak maksymalne wartości zużycia CPU i RAMu były podobne dla każdego typu, tak średnia CPU różniła się znacząco między typami. Na tej podstawie wyciągnięto kilka wniosków co do wyboru algorytmów porównywania szeregów czasowych.

- Fakt pojawiania się wartości odstających może negatywnie wpływać na rozróżnianie UDFów za pomocą odległości Euklidesowej. Z drugiej strony to, że długość szeregów czasowych potrafi przynajmniej częściowo determinować typ funkcji sprawia, że jest to algorytm warty rozważenia. Problematyczne będą jednak przypadki porównywania dwóch identycznych funkcji, ale użytych na różnych rozmiarach danych.

- Algorytm Dynamic Time Warping jest stworzony do radzenia sobie ze skalowaniem czasu. Jest to pozytywny efekt, gdy patrzy się na to, że celem jest porównanie szeregów czasowych, które niekoniecznie mają ten sam rozmiar. Z drugiej strony czas wykonania jest tym co wyraźnie odróżnia część algorytmów od siebie, rezygnacja z tej informacji niekoniecznie jest dobrym rozwiązaniem.

- Algorytm Landmark Similarity również radzi sobie ze skalowaniem czasu, ale też skalowaniem amplitudy. Zwraca uwagę niemal wyłącznie na kształt wykresu. Jest to problematyczne w przypadku omawianym w tej pracy, gdyż zarówno amplituda jak i długość szeregu przekazuje dużo informacji o typie funkcji.

Rozdział 4

Oprogramowanie i przygotowanie danych

Tworząc niniejszą prace wykorzystano język Python i dostępne w nim implementacje algorytmów badających podobieństwo między szeregami czasowymi oraz klasyfikujących te szeregi. W projekcie użyto Python'a w wersji 3.9.x. Chcąc umożliwić zainteresowanym osobą powtórzenie eksperymentów, poszczególne etapy przetwarzania zostały przygotowane przy pomocy notatników Jupiterowych (ang. *Jupyter Notebooks*). Umożliwiają one umieszczenie kodu w komórkach (ang. *cells*) oraz udokumentowanie go przy pomocy języka znacznikowego *Markdown*.

Wszystkie surowe dane znajdują się w repozytorium [15] w folderze "Raw", a wraz z nimi można znaleźć notatniki w folderze "Notebooks". Każdy z nich rozpoczyna się od liczby, przykładowo "1_data_processing". Powinny one zostać uruchomione w kolejności rosnącej w celu uzyskania poprawnie przetworzonych danych oraz uzyskanych w pracy wyników eksperymentów. Dodatkowo, repozytorium zawiera plik "requirements.yml", którego zawartością są wszystkie potrzebne zależności, konieczne do poprawnego wykonania się kodu.

W poniższym rozdziale poza opisem poszczególnych etapów przygotowywania danych, pojawi się zestawienie bibliotek, które zostały sprawdzone i wybrane na potrzeby części eksperymentalnej. Kryteriami wyboru biblioteki jest ich stabilność, wiarygodność oraz użyteczność w połączeniu z danymi przedstawiającymi szeregi czasowe. Stabilność oznacza, czy implementacje zawarte w bibliotece wykonują się bez błędów, czyli nie pojawiają się niespodziewane wyjątki. Wiarygodność sprowadza się do tego, czy zwracane wyniki są powtarzalne dla tych samych parametrów przekazywanych podczas uruchomienia metody. Użyteczność to cecha, która oznacza czy zawartość biblioteki (zaimplementowane algorytmy) są przydatne do przeprowadzanych badań w niniejszej pracy.

4.1 Wykorzystane biblioteki do eksperymentów

4.1.1 Biblioteki implementujące algorytmy obliczające podobieństwo szeregów czasowych

SciPy

`SciPy` [16] zawiera zbiór algorytmów do obliczeń z różnych dziedzin matematyki. Biblioteka zbudowana jest na paczce **NumPy** oraz jest podzielona tematycznie na klika działów. Moduł `scipy.spatial` skupia się na algorytmach dotyczących danych przestrzennych. Jednym z zaimplementowanych algorytmów jest odległość euklidesowa między jednowymiarowymi tablicami. Ta funkcja to `distance.euclidean()` przyjmująca poniższe parametry:

- `u`: szereg czasowy 1 - Dane wejściowe (pierwsza tablica)

- `v`: szereg czasowy 2 - Dane wejściowe (druga tablica)
- `w`: parametr opcjonalny, zawierający wagi dla każdej wartości w `u` i `v`. Jeżeli wynosi None to każda wartość ma równą wagę 1.0

dtw-python

Dtw-python [17] dostarcza implementacje algorytmu Dynamic Time Warping. Poza zwróceniem odległości między szeregami, umożliwia on utworzenie wykresu przedstawiającego ścieżkę powstałą w wyniku działania algorytmu. Biblioteka daje możliwość użycia algorytmu DTW z różnymi wzorcami kroku - zarówno symetrycznymi, jak i asymetrycznymi. Krok symetryczny dopasowuje nielimitowaną liczbę elementów pierwszego szeregu do jednego elementu drugiego szeregu i vice versa. Krok asymetryczny oznacza, że każdy element szeregu może być dopasowany tylko raz. Możliwe jest również wybranie miary odległości użytej do obliczenia kosztu. Funkcja dtw() zwraca obiekt, który zawiera między innymi wyliczoną odległość i wyliczoną odległość znormalizowaną.
Najważniejsze parametry funkcji dtw():

- `x`: szereg czasowy 1
- `y`: szereg czasowy 2
- `dist_method` : miara odległości użyta do obliczenia wartości kosztu, np. euclidean, jaccard, correlation, cosine; domyślną wartością jest euclidean
- `step_pattern`: wzorzec kroku, np. symmetric1, symmetric2, asymmetric, rabinerJuangStepPattern

Odległość znormalizowana wyliczana jest przy użyciu funkcji normalizującej, która zależy od wartości step pattern. Przykładowo, dla symmetric2 jest to podzielenie odległości przez N + M, gdzie N jest liczbą wartości pierwszego szeregu czasowego, a M jest liczbą wartości drugiego szeregu czasowego. Nie każdy step pattern umożliwia normalizacje.

dtaidistance

Kolejna biblioteka implementująca algorytm Dynamic Time Warping to dtaidistance [18]. Służy do mierzenia dystansów między szeregami czasowymi i jest używana przez DTAI Research Group. Oferuje zarówno implementację w czystym Pythonie, jak i w C. Jest kompatybilna z NumPy i Pandas.
Funkcja distance() zwraca odległość między podanymi szeregami czasowymi. Jej najważniejsze parametry:

- `s1`: szereg czasowy 1
- `s2`: szereg czasowy 2

tslearn.metrics lcss_path

Biblioteka **tslearn** [19] zawiera narzędzia do uczenia maszynowego używając danych zawierających szeregi czasowe. Znajdują się w niej również inne metryki związane z szeregami czasowymi. Tu wykorzystano funkcję zwracającą wynik algorytmu Longest Common Subsequence - odległość oraz ścieżkę. Przyjmuje ona następujące parametry:

- `s1`: szereg czasowy 1
- `s2`: szereg czasowy 2
- `eps`: próg maksymalnego dopasowania

4.1.2 Biblioteki implementujące algorytmy klasyfikujące szeregi czasowe

Do przeprowadzenia eksperymentów, potrzebne są zaimplementowane klasyfikatory, które przyjmują jako dane wejściowe szeregi czasowe. Tak jak wspomniano w rozdziale 2., istnieją algorytmy, które specjalizują się w klasyfikacji szeregów czasowych. Ostatecznie zdecydowano się na sprawdzenie dwóch bibliotek, a mianowicie wcześniej wspomniany **tslearn** [19] oraz **sktime** [20], [21].

tslearn

W poprzedniej części ta biblioteka została wspomniana, gdy omawiano implementacje metryk dotyczących szeregów czasowych. Poza tym znajdują się w niej klasyfikatory, które zostały przystosowane do tego typu danych. Bazują one na znanej w świecie uczenia maszynowego bibliotece **scikit-learn**. Twórcy zaimplementowali również metody, które umożliwiają wczytanie zbioru danych w prosty sposób. Na przykład można przekształcić tablice wielowymiarowe (np. takie zawierające kolejno zużycie CPU oraz RAM), w format działający z tą właśnie biblioteką. Dodatkowo, umożliwia konwersje formatu danych w obie strony dla innych bibliotek przetwarzających szeregi czasowe. Przykładowe biblioteki to **sktime**, **tsfresh** i wiele innych[1].

W module można wyróżnić trzy implementacje dotyczące klasyfikacji:

- **KNN classifier**[2] - implementacja algorytmu K najbliższych sąsiadów przystosowana do szeregów czasowych. Tak jak wspomniano w rozdziale 2., do potrzeb tej pracy, najbardziej użyteczna będzie wersja tego algorytmu w połączeniu z metryką Dynamic Time Warping. Jest to domyślna funkcja odległości użyta w tej implementacji. Ta wersja KNN może zostać użyta do problemu wielowymiarowego.
- **TimeSeriesSVC**[3] - przystosowany do klasyfikacji szeregów czasowych algorytm Support vector machine.
- **LearningShapelets**[4] - jedna z specjalistycznych metod służąca do klasyfikacji szeregów, którą autorzy biblioteki oparli na artykule [22]. Jest to również jeden z algorytmów zawartych w metodzie HIVECOTEV2, opisanej w rozdziale 2.

Ten moduł jednak nie udostępnia takich metod jak ROCKET czy HIVECOTEV2, więc z jego pomocą nie ma możliwości zbadania działania tych podejść.

sktime

sktime to biblioteka, która udostępnia środowisko do analizy danych w postaci szeregów czasowych. Można w niej znaleźć wszelakiej maści klasyfikatory, regresory, algorytmy grupujące czy

[1] https://tslearn.readthedocs.io/en/stable/gen_modules/tslearn.utils.html#module-tslearn.utils
[2] https://tslearn.readthedocs.io/en/stable/gen_modules/neighbors/tslearn.neighbors.KNeighborsTimeSeriesClassifier.html
[3] https://tslearn.readthedocs.io/en/stable/gen_modules/svm/tslearn.svm.TimeSeriesSVC.html#tslearn.svm.TimeSeriesSVC
[4] https://tslearn.readthedocs.io/en/stable/gen_modules/shapelets/tslearn.shapelets.LearningShapelets.html

prognozujące przebiegi czasowe. Ponownie mamy do czynienia z modułem bazującym na **scikit-learn**, co sprawia, że jest kompatybilny z narzędziami dostępnymi w tej bibliotece. Autorzy zaimplementowali również metody tworzące zestawy danych, które działają wraz z udostępnionymi algorytmami. Dodatkową zaletą biblioteki **sktime** jest dostęp do interfejsu służącego do testowania algorytmów na różnych zbiorach danych oraz tworzenia rankingów. W bibliotece można znaleźć następujące algorytmy służące do klasyfikacji szeregów czasowych:

- **KNN classifier**[5] - biblioteka udostępnia przystosowany do szeregów czasowych klasyfikator K najbliższych sąsiadów. Pozwala on na wykorzystanie różnych metryk obliczających podobieństwo szeregów czasowych, w tym algorytm DTW. Podobnie jak w przypadku **tslearn**, można go użyć dla problemu wielowymiarowego.
- **HIVECOTEV2**[6] - jeden z dostępnych wyspecjalizowanych metod klasyfikacji szeregów czasowych. Ta wersja jest zgodna z specyfikacją opisaną w rozdziale 2., zatem badany zestaw danych może zostać wykorzystany bez przeszkód.
- **ROCKET**[7] - transformacja losowymi konwolucyjnymi kernelami opisana w rozdziale 2., zawarta wraz z regresją grzbietową. Wersja ta udostępnia możliwość wykorzystania danych wielowymiarowych oraz inne metody transformacji (multi-rocket, mini-rocket), które należy zbadać pod kątem trafności predykcji.

Poza klasyfikatorami opisanymi powyżej, zaimplementowane zostało również wiele innych specjalistycznych algorytmów klasyfikujących szeregi czasowe. Dodatkowo, jeżeli dana metoda nie wspiera bezpośrednio danych wielowymiarowych, możliwe jest przygotowanie transformacji danych w taki sposób, aby traktowane były one jako jednowymiarowe (przykładowo przyłączając jeden z wymiarów po pierwszym z nich.).

Wybór biblioteki do eksperymentów

Aby zadecydować, która z bibliotek powinna zostać użyta w pracy, podjęto próby ich uruchomienia z wykorzystywanym w pracy zbiorem danych. **tslearn** jak i **sktime** wymagają pewnego wypełnienia danych w celu uzyskania ich równej długości (za pomocą zer lub wartości pustych), co jest wymagane, by algorytmy działały prawidłowo. Biblioteka **sktime** implementuje więcej specjalistycznych algorytmów od **tslearn**, które umożliwiają użycie danych wielowymiarowych bez ich dodatkowego przetwarzania. Klasyfikator SVC z biblioteki **tslearn** zwraca błędy, których nie udało się rozwiązać. Wynikają one z błędu w kodzie implementującym klasyfikatory SVC. Zbyt długie szeregi (dłuższe niż około 300 obserwacji) nie mogą zostać poprawnie przetworzone (próba uruchomienia kończy się błędem po długim przetwarzaniu). Sprawia to, że nie można go wykorzystać w badaniach. Z tych powodów do działań związanych z klasyfikacją szeregów czasowych wybrano bibliotekę **sktime**. Umożliwia ona sprawdzenie algorytmów opisanych w rozdziale 2., czyli KNN-DTW, ROCKET oraz HIVECOTEV2.

[5] `https://www.sktime.org/en/latest/api_reference/auto_generated/sktime.classification.distance_based.KNeighborsTimeSeriesClassifier.html`

[6] `https://www.sktime.org/en/latest/api_reference/auto_generated/sktime.classification.hybrid.HIVECOTEV2.html`

[7] `https://www.sktime.org/en/latest/api_reference/auto_generated/sktime.transformations.panel.rocket.Rocket.html?highlight=ROCKET`

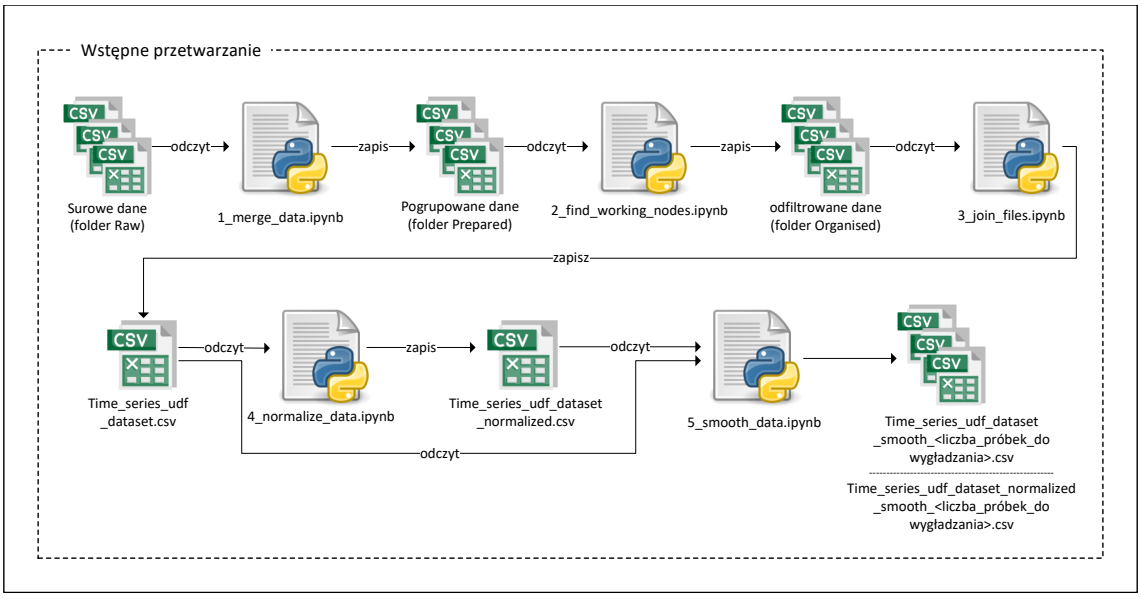
4.2 Przetwarzanie i wizualizacja danych

4.2.1 Przepływ przetwarzania danych

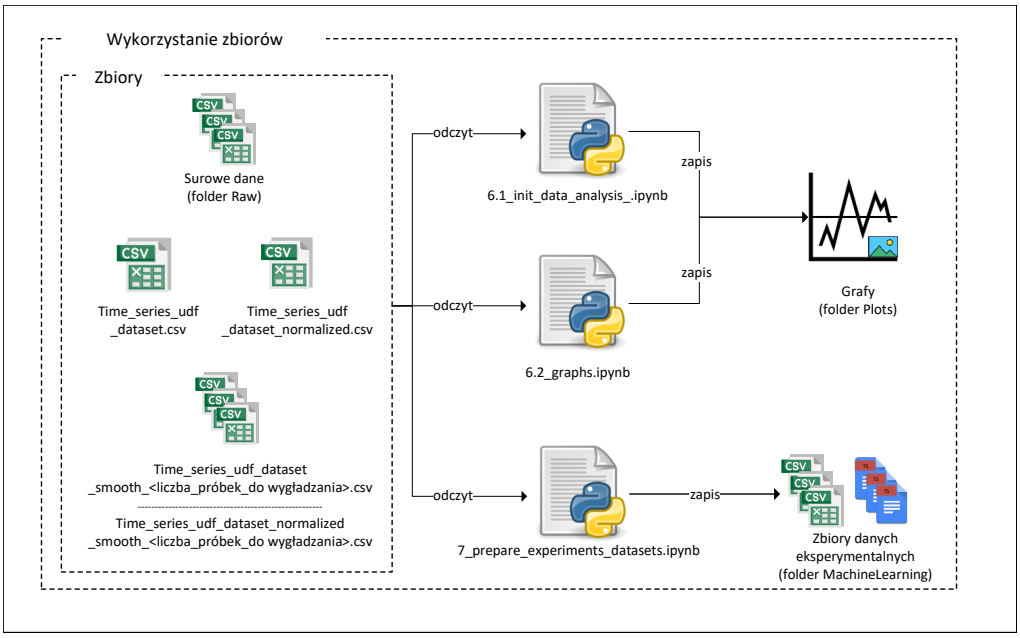
Do przeprowadzenia wstępnego przetwarzania danych wykorzystano *Jupiter Notebooks*. Zawartość poszczególnych notatników to skrypty przygotowujące dane do eksperymentów. Każdy z nich obsługuje inny etap. Poniżej znajduje się lista notatników dotyczących przetwarzania oraz wizualizacji danych wraz z krótkim opisem:

- 1_merge_data.ipynb - odpowiada za wczytanie surowych danych z folderu *Raw* oraz zgrupowanie ich w ramach znaczników czasowych. Następnie w ramach utworzonych grup, sumuje wartości zużycia zasobów i zapisuje przygotowane dane w folderze *Prepared*.
- 2_find_working_nodes.ipynb - wykonuje odpowiednią filtrację w celu wydobycia szeregów z węzłów pracujących. Dane pochodzą z folderu *Prepared*. Tak odfiltrowane pliki CSV trafiają do folderu *Organised*.
- 3_join_files.ipynb - notatnik, który doprowadza dane do formy jaką przyjęto jako ostateczną, wygodną w dalszych eksperymentach. Łączy on wszystkie pliki z folderu *Organised* w jeden zbiorczy plik CSV znajdujący się w tym samym folderze.
- 4_normalize_data.ipynb - wykonuje normalizacje danych przy pomocy podejścia **Min-Max**. Plik utworzony po normalizacji znajduje się w tym samym miejscu co oryginalny zbiorczy plik.
- 5_smooth_data.ipynb - dokonuje dodatkowe wygładzenie (ang. *smoothing*) danych przed i po normalizacji. Wyniki zapisane są w plikach w tym samym miejscu co oryginalny plik, z dopiskiem o wygładzeniu oraz liczbą próbek wziętych pod uwagę podczas wygładzania.
- 6.1_init_data_analysis.ipynb - notatnik służący do przeanalizowania danych wejściowych. Grafy w nim utworzone zostały wykorzystane w rozdziale 3.2.
- 6.2_graphs.ipynb - notatnik generujący grafy użyte w rozdziale 4. Wszystkie wyjściowe pliki zostają zapisane w folderze *Plots*.
- 7_prepare_experiments_datasets.ipynb - notatnik wykorzystujący pliki utworzone w ramach poprzednich notatników. Tworzy on zestawy danych, które mają posłużyć do wykonania eksperymentów w niniejszej pracy. Są to dane w formacie CSV oraz specjalnym formacie ts, działającym z biblioteką **sktime**.

Poszczególne etapy opisane są w dalszej części pracy. Na rysunkach 4.1 oraz 4.2 znajdują się schematy przedstawiające poglądowy przepływ danych (ang. *Data Processing Pipeline*). Pokazują one interakcje notatników wymienionych powyżej.



Rysunek 4.1: Przepływ wstępnego przetwarzania danych



Rysunek 4.2: Notatniki wykorzystujące zbiory z wczesnego przetwarzania.

4.2.2 Uporządkowanie danych

Jak sama nazwa wskazuje, w procesie uczenia maszynowego kluczowe są dane, za pomocą których algorytmy mogą się "uczyć". Na podstawie tego stwierdzenia surowe dane, które udostępnili autorzy pracy [5], należy wpierw odpowiednio przygotować przed ich użyciem w eksperymentach. W tej sekcji opisano uporządkowanie danych, ich wstępne przetwarzanie oraz proces normalizacji tak przygotowanego zbioru.

Pierwszym etapem przetwarzania danych było ich uporządkowanie i zapisanie w formacie odpowiednim dla dalszych eksperymentów. W pierwszej kolejności konieczne było pozbycie się podziału na procesy. W tym celu została wykorzystana biblioteka **Pandas** [23], która jest często używana w przetwarzaniu danych. Dane z rozszerzeniem CSV z folderu *Raw* zostały kolejno odczytane, a następnie wpisy pogrupowano po znacznikach czasowych z kolumny "timestamp". Wartości zuży-

cia zasobów (kolumny CPU oraz RAM) zsumowano w ramach jednej grupy (znacznika czasowego). Dodatkowo, usunięto identyfikatory procesów, gdyż nie są one potrzebne do dalszego przetwarzania. Kolumna "timestamp" została zastąpiona kolumną "epoch", zawierająca liczbę sekund (z dokładnością do 16 miejsc po przecinku) która minęła od uruchomienia funkcji. Otrzymane zbiory zostały zapisane w takiej samej strukturze jak wcześniej, ale w osobnym nadfolderze *Prepared*. Przykładowy plik wynikowy pokazany jest w tabeli 4.1.

Tablica 4.1: Początek pliku aggregation/avgNetProfitGroupedBySoldDate/CPU-data-vol-1GB-5nodes/source-data/node-11/07_08_2021_23_43_34.csv w folderze Prepared na repozytorium [15]

epoch	CPU	RAM
0.0	206.7	4.8
0.15627002716064453	220.0	5.0
0.31401014328002930	133.3	5.2
0.47159314155578613	146.7	5.3
0.6280381679534912	126.7	5.4
0.7846581935882568	160.0	5.5
0.9419929981231690	206.7	5.5

W dalszym ciągu pozostaje kwestia zawartości poszczególnych folderów węzłów. Dla każdego uruchomienia funkcji są dostępne przebiegi o zużyciu w każdym węźle, zamiast ogólnie dla UDF'a. Dodatkowo, bazując na pracy pilotażowej [5], wiadomo że nie każdy węzeł aktywnie brał udział w obliczeniach. Jeden z węzłów był zawsze używany jako master, którego zadaniem było zarządzanie wykonywaniem zadań na klastrach pracujących (ang. *workers*). Jeżeli master nie nadał zadania każdemu z klastrów pracujących, to taki węzeł pozostawał bezczynny, z wyjątkiem okazyjnego wysyłania informacji o stanie (tzn. czy jest wciąż aktywny). Z tego powodu następnym krokiem było odfiltrowanie wszystkich węzłów bezczynnych oraz mastera i zachowanie jedynie tych, które wykonywały obliczenia.

Założono, że master, ze względu na to, że nie wykonuje pracy na danych, nie powinien wykorzystywać zbyt dużo pamięci RAM. W rzeczy samej dla każdego eksperymentu można było znaleźć jeden węzeł, którego średnie zużycie RAM było wyraźnie mniejsze. Eksperymentując, ustalono próg 2% - odrzucono wszystkie węzły, dla których średnie zużycie RAM było poniżej tego progu. Następnym krokiem było znalezienie i wykluczenie węzłów niepracujących. Jako, że takie węzły wysyłały tylko okazyjne informacje o tym, że wciąż są aktywne to różnice między zużyciem CPU dla nich i dla węzłów pracujących również były wyraźnie widoczne. Dokładnie to było widać zwłaszcza w kilku początkowych pomiarach, w których węzły pracujące charakteryzowały się wysokim zużyciem CPU, co było dokładnie opisane i wytłumaczone w pracy pilotażowej [5]. Z tego względu odrzucono węzły, których początkowe pomiary były niskie. Po odfiltrowaniu zbędnych szeregów, otrzymane zbiory zostały zapisane w takiej samej strukturze jak uprzednio, ale w osobnym nadfolderze *Organised*.

Aby ułatwić dalsze przetwarzanie danych, należało uzyskać jak najmniej plików wejściowych. W tym celu dane zostały połączone w jeden zbiorczy plik. Inspirując się przykładowym zbiorem danych wykorzystywanym do uczenia maszynowego za pomocą danych typu Time Series[8], utworzony został plik CSV zawierający w sobie szeregi dla wszystkich typów funkcji. Został zapisany pod nazwą "Time_series_udf_dataset.csv" w folderze *Organised* z następującym schematem.

- snapshot - identyfikator pojedynczego przebiegu czasowego
- label - opis typu funkcji, np "aggregation"

[8] `https://github.com/trdinger/Public-NN-Rocket-Code/blob/main/sample.csv`

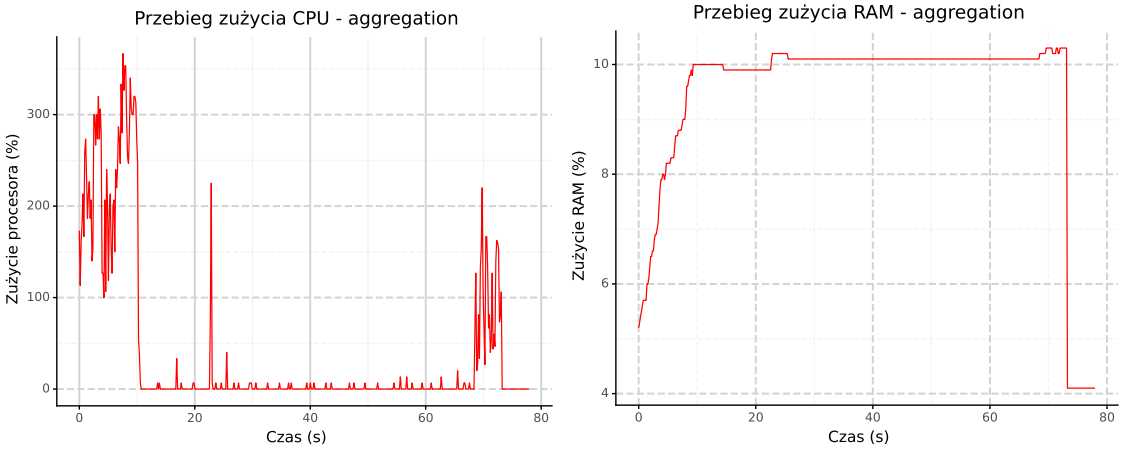
- udf - nazwa, z którego pochodzi danych przebieg czasowy
- epoch - liczba sekund, która minęła od uruchomienia funkcji z dokładnością do 16 miejsc po przecinku
- CPU - procent zużycia procesora przez dany proces (dla zadań wielordzeniowych może przekroczyć 100%)
- RAM - procent zużycia pamięci przez dany proces
- size - rozmiar danych przetwarzanych w danym szeregu (w GB)

W tablicy 4.2 przestawiono przykładowe kilka wierszy z tak utworzonego pliku:

Tablica 4.2: Początek pliku Time_series_udf_dataset.csv z folderu "Organised" na repozytorium [15]

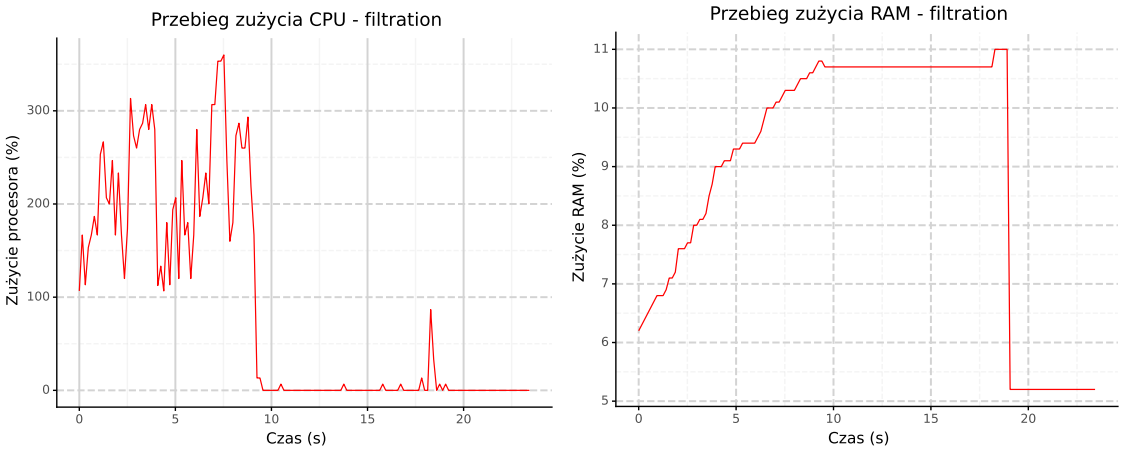
snapshot	label	udf	epoch	CPU	RAM	size
0	aggregation	avgNetProfit- -GroupedBySoldDate	0.0	206.7	4.8	1
0	aggregation	avgNetProfit- -GroupedBySoldDate	0.1562700271606445	220.0	5.0	1
0	aggregation	avgNetProfit- -GroupedBySoldDate	0.3140101432800293	133.3	5.19	1
0	aggregation	avgNetProfit- -GroupedBySoldDate	0.4715931415557861	146.7	5.3	1
0	aggregation	avgNetProfit- -GroupedBySoldDate	0.6280381679534912	126.7	5.4	1
0	aggregation	avgNetProfit- -GroupedBySoldDate	0.7846581935882568	160.0	5.5	1

Zebrane w ten sposób dane mogą być dalej przetwarzane. W zbiorze można wyróżnić pięć typów funkcji zależnych od rodzaju przetwarzania, które opisano w rozdziale 3. Na rysunkach od 4.3 do 4.7 przedstawiono przykładowe przebiegi czasowe na wykresach, gdzie zawarto podział CPU i RAM dla każdego z typów UDF w pliku utworzonym według założeń.



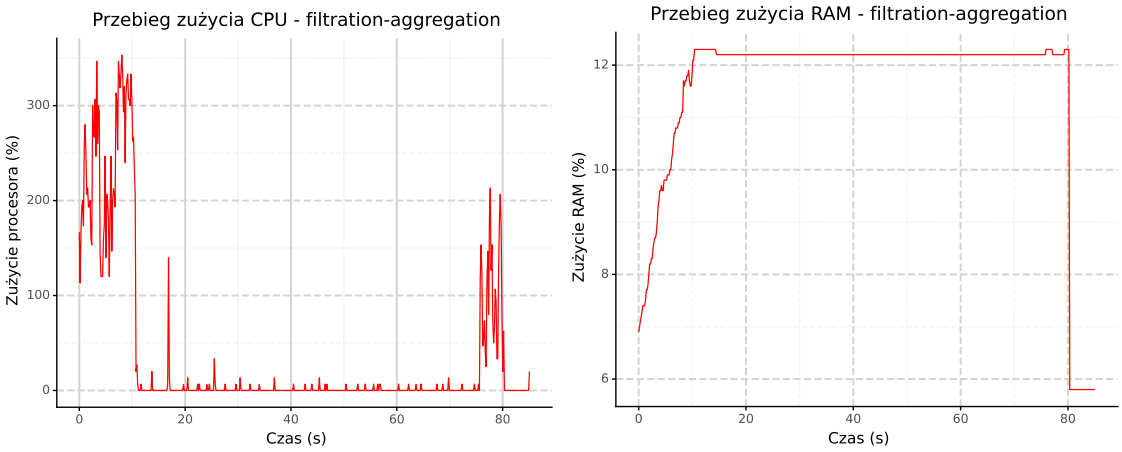
(a) CPU (b) RAM

Rysunek 4.3: Przykładowy przebieg zużycia zasobów dla agregacji (snapshot = 1)



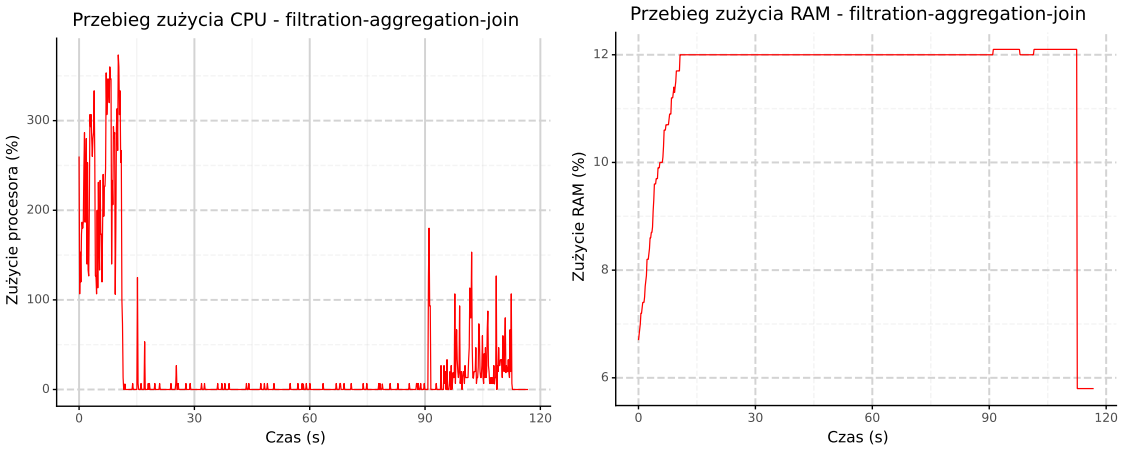
(A) CPU (B) RAM

Rysunek 4.4: Przykładowy przebieg zużycia zasobów dla filtracji (snapshot = 1)



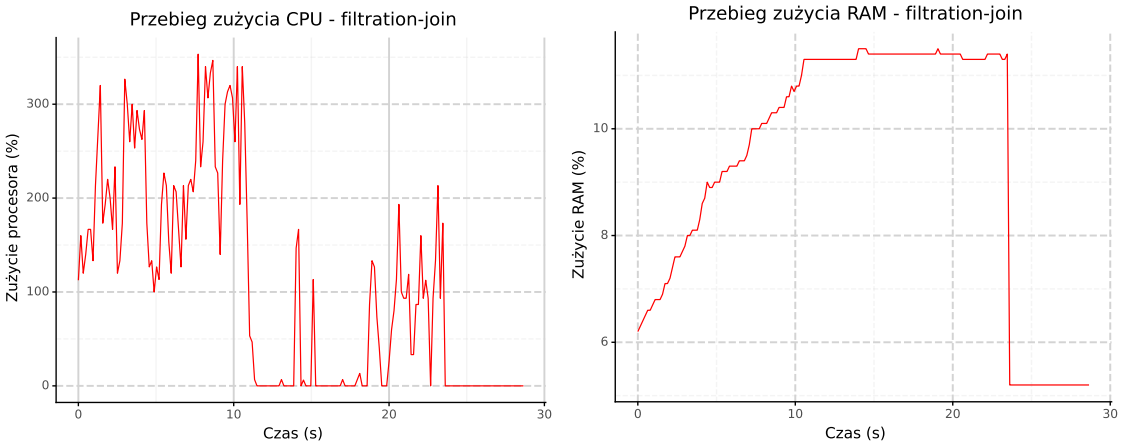
(A) CPU (B) RAM

Rysunek 4.5: Przykładowy przebieg zużycia zasobów dla filtracjo-agregacji (snapshot = 1)



(A) CPU (B) RAM

Rysunek 4.6: Przykładowy przebieg zużycia zasobów dla filtracjo-agregacji z połączeniem (snapshot = 1)

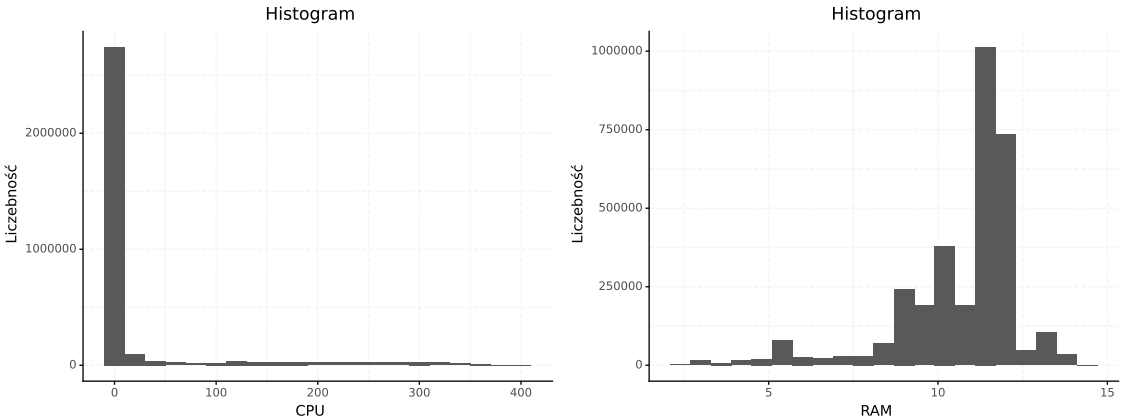


(A) CPU (B) RAM

Rysunek 4.7: Przykładowy przebieg zużycia zasobów dla filtracji z połączeniem (snapshot = 1)

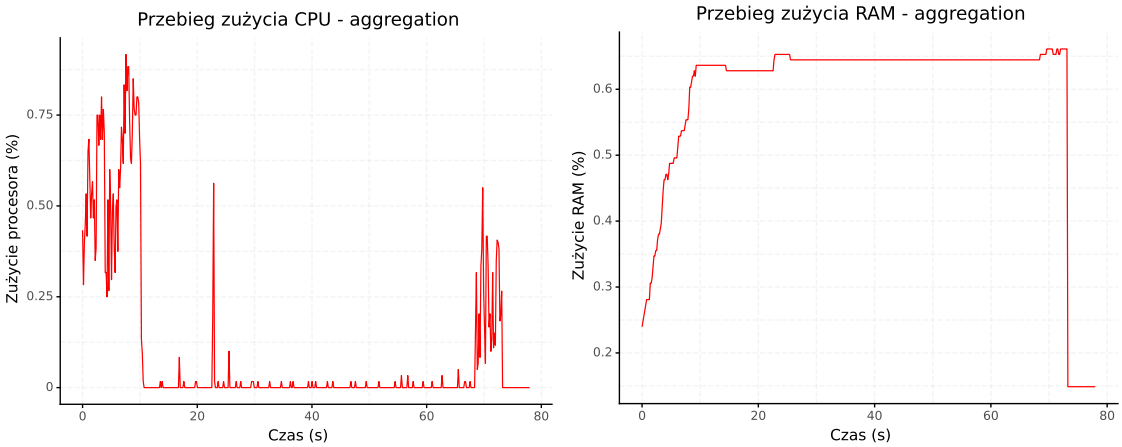
4.2.3 Normalizowanie danych

Przygotowując dane do analizy częstym działaniem jest ich normalizacja. W rozdziale 2.3 opisano sposoby normalizacji. Badając używany zbiór danych, można stwierdzić że nie ma on rozkładu normalnego. Można to zauważyć na wykresach 4.8. Bazując na teorii, że dla danych tego typu lepiej działa technika **Min-Max**, to właśnie ona zostanie wykorzystana podczas działań na zbiorze na potrzeby niniejszej pracy.



(A) CPU (B) RAM

Rysunek 4.8: rozkład danych dla agregacji



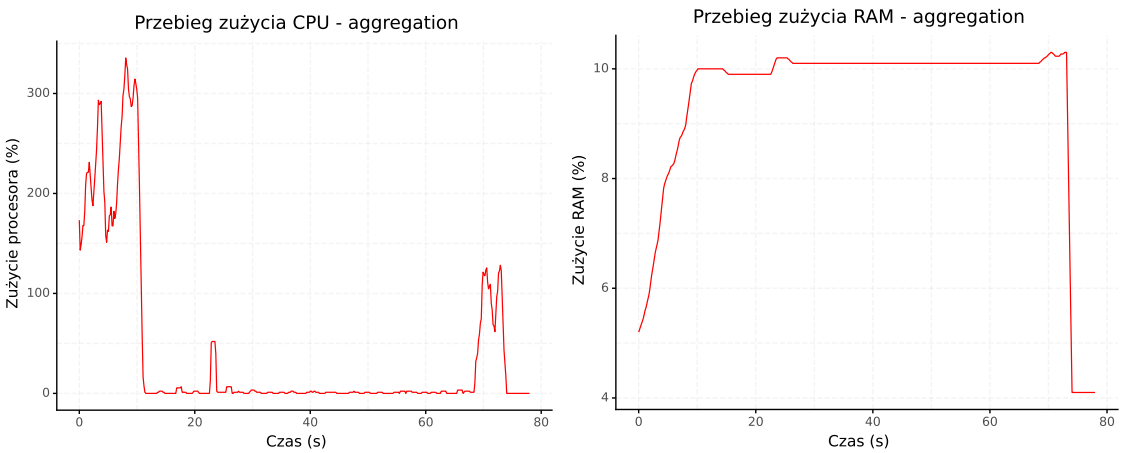
(A) CPU (B) RAM

Rysunek 4.9: Przykładowy znormalizowany wykres zużycia zasobów dla agregacji (snapshot = 1)

Wykresy 4.9 wyraźnie wskazują na to, że po wykonaniu normalizacji kształt przebiegów się nie zmienił (porównując z 4.3a), jednak ich amplitudy zmalały. W ten sposób przygotowane dane zostaną wykorzystane w eksperymentach. Będą one sprawdzały jakość algorytmów przed i po normalizacji. Plik zawierający znormalizowane dane został zapisany pod nazwą "Time_series_udf-_dataset_normalized.csv", w folderze "Organised"

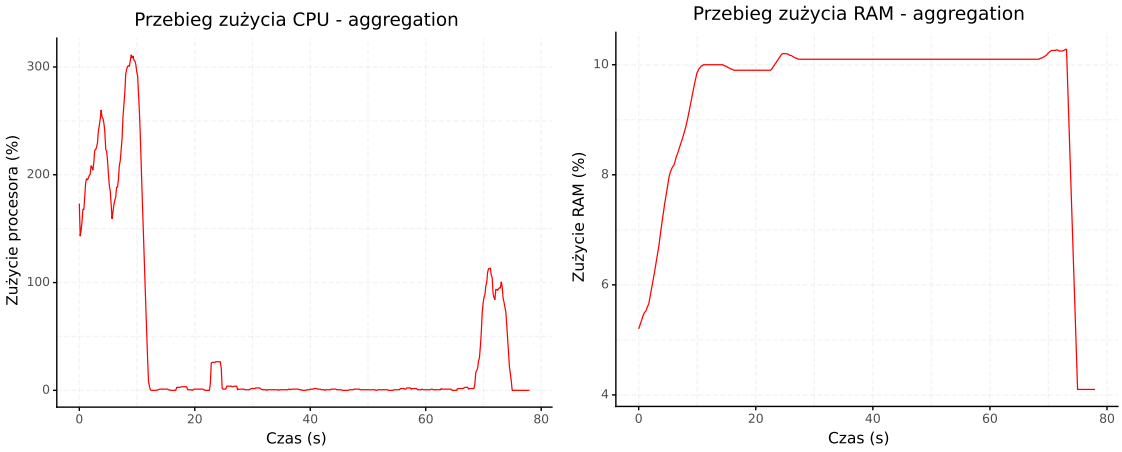
4.2.4 Wygładzenie danych

Ostatnią transformacją, która została przeprowadzona na danych jest ich wygładzenie. Na rysunkach 4.3 do 4.7 można zauważyć, że wykresy CPU są bardzo ostre i mogłyby zyskać na wygładzeniu krawędzi. Operacja ta została wykonana poprzez wykorzystanie średniej w oknie przesuwnym. Okno przesuwne wykorzystuje liczbę próbek zależną od tego, jak duży okres czasu ma zostać wzięty pod uwagę. Próbki są rejestrowane w odstępach co ok. $\frac{1}{15}$ sekundy. Zaokrąglając, sześć próbek jest zatem równa jednej sekundzie. Poniżej znajdują się wykresy przestawiające wygląd szeregów po wygładzeniu dla różnej liczby próbek na przykładzie funkcji agregującej.



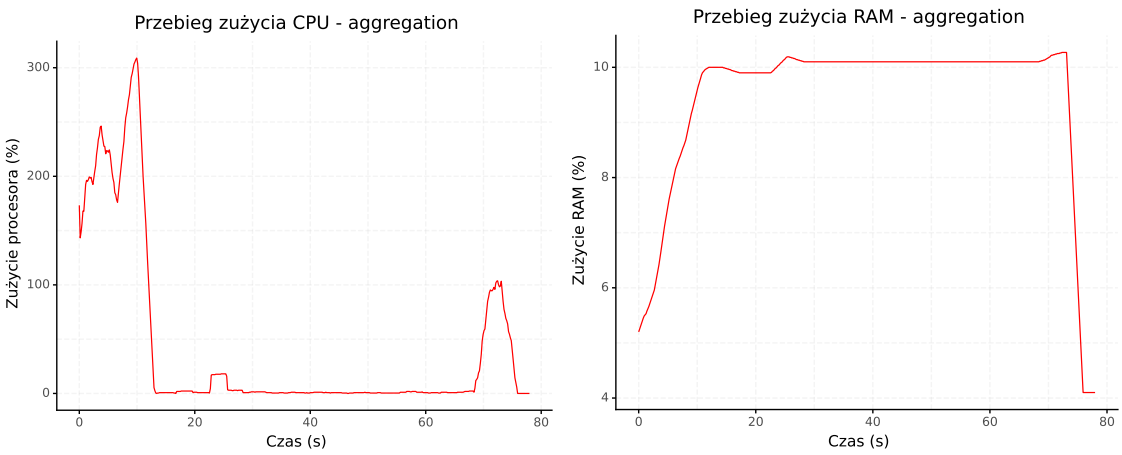
(A) CPU (B) RAM

Rysunek 4.10: Przykładowy wygładzony wykres zużycia zasobów dla agregacji (snapshot = 1, okno 1 sekundowe)



(A) CPU (B) RAM

Rysunek 4.11: Przykładowy wygładzony wykres zużycia zasobów dla agregacji (snapshot = 1, okno 2 sekundowe)



(A) CPU (B) RAM

Rysunek 4.12: Przykładowy wygładzony wykres zużycia zasobów dla agregacji (snapshot = 1, okno 3 sekundowe)

Po wykonaniu transformacji można wysnuć wniosek, że wykresy są bardziej stabilne (duże skoki wartości w małym przedziale czasowym są rzadsze). Rysunki od 4.10 do 4.12 pokazują wyniki dla kolejno 6, 12 i 18 próbek (czyli 1, 2, 3 sekundy) w oknie. Ostatecznie, aby nie doprowadzić do zbytniego wygładzenia danych, zostanie wykorzystany zbiór z oknem jednosekundowym. Zbyt duże okno doprowadziłoby to utraty informacji, która mogłaby być kluczowa przy badaniu podobieństwa. Przykładowo, na rysunku CPU 4.12 widoczne jest w porównaniu z rysunkiem 4.10 wyraźnie zmniejszony szczyt w okolicy 25 sekundy.

Pliki wygładzone można wykorzystać przed i po normalizacji, jeżeli eksperymenty by tego wymagały. Wszystkie pliki zawierające te dane mają następującą strukturę nazwy: "Time_series-_udf_dataset_smooth_<liczba_próbek_do_wygładzania>.csv" i analogicznie dla plików znormalizowanych "Time_series_udf_dataset_normalized_smooth_<liczba_próbek_do_wygładzania>.csv"

4.2.5 Przygotowanie zbioru danych eksperymentalnych

W celu przygotowania zbioru danych do przeprowadzenia uczenia maszynowego i innych eksperymentów, należy podzielić go na tzw. zbiór testowy (ang. *test*) i treningowy (ang. *train*). Przed

podziałem zbiór ten wymaga transformacji, aby dopasować jego format do wybranej biblioteki. Eksperymenty zostaną podzielone na dwa problemy. Pierwszy z nich to **zbadanie podobieństwa oraz klasyfikacja szeregów w ramach jednego typu funkcji** z podziałem na klasy zależne **od wielkości wolumenu danych**. W tym przypadku badane klasy to 1GB oraz 2GB. Takie eksperymenty mają na celu sprawdzenie, czy niezależnie od rozmiaru danych funkcje w ramach jednego typu mają podobne szeregi.

Drugim zaś problemem jest **zbadanie podobieństwa i klasyfikacja między typami funkcji**. Wtedy klasy będą pochodzić z każdej z pięciu badanych kategorii **bez względu na rozmiar przetwarzanych danych**.

W obu przypadkach, chcąc uniknąć problemu niezbalansowanych danych jak i zbyt długiego przetwarzania, liczebności zbiorów zostaną ograniczone. W przypadku problemu podobieństwa między typami UDF, ograniczono liczność każdej z klas do liczby szeregów klasy "filtering". Spowodowane to jest faktem, że klasa "flitering" zawiera jedynie trzysta szeregów. W celu zachowania pewnej losowości danych, poszczególne próbki będą wybierane poprzez losowanie z zbioru identyfikatorów przebiegów (w pliku pole "snapshot").

Dla eksperymentów badających podobieństwo szeregów czasowych, o różnych wolumenach przetwarzanych danych również dokonano podobnego ograniczenia liczebności zbiorów, w ramach każdego typu funkcji. Dla funkcji typu "Filtering" dostępnych jest jedynie trzysta szeregów (po sto pięćdziesiąt szeregów dla klas 1GB oraz 2GB), więc w tym przypadku wykorzystane zostały wszystkie dostępne dane. Dla reszty typów UDF, w celu zachowania spójności z eksperymentami dotyczącymi badania rozróżnialności typów, każda z klas (1GB i 2GB) zawiera po trzysta losowych szeregów.

Po zebraniu połączonego zbioru zachodzi potrzeba dokonania transformacji w taki sposób, żeby dane pasowały do użytej biblioteki. Wybrany **sktime** wymaga, aby przebiegi zawierały równą liczbę obserwacji w każdym szeregu dla poprawnego działania algorytmów. W tym celu konieczne było wyrównanie długości szeregów do najdłuższego z nich. Krótsze szeregi zostały zatem wypełnione za pomocą zer. Spowodowało to zachowanie pewnej zgodności z faktem, że w okresie dopełnianym maszyna nie wykonywała żadnych działań. Jednym z argumentów przemawiającym za wyborem biblioteki **sklearn** były udostępnione w niej funkcje, które transformują przygotowane dane do formatu, działającego z zaimplementowanymi klasyfikatorami. Metody te zostały wykorzystane w tym etapie przetwarzania, co pozwoliło na utworzenie szeregów w formacie zgodnym z wybranymi implementacjami algorytmów.

Ostatnim krokiem jest podzielenie zbioru na rozłączne podzbiory testowe i treningowe. **sktime** opiera swoje klasyfikatory na bibliotece sklearn, która zawiera w sobie wiele narzędzi służących do przygotowywania danych do procesu uczenia maszynowego. W używanej bibliotece znajdują się metody dzielące dane na wspomniane zbiory treningowe i testowe z parametrami określającymi liczność oraz umożliwiającymi kontrole nad losowością. W tym przypadku zbiór testowy zawiera $\frac{1}{3}$ przygotowanych danych, tak więc treningowy zawiera $\frac{2}{3}$ z nich. Generatory liczb losowych posiadają parametr nazwany *seed*. Przy każdym wywołaniu generatora liczb losowych otrzymuje się liczbę opartą na aktualnie ustawionej wartości tego parametru. Dzięki temu przy każdym uruchomieniu metody dzielącej zbiory, uzyskamy tak samo podzielone zestawy danych. W tym przypadku parametr ustawiono na liczbę czterdzieści dwa.

Aby ułatwić przeprowadzanie eksperymentów, każdy z utworzonych zbiorów został zapisany w odpowiednich folderach w dwóch formatach: ts oraz oryginalnym CSV. Rozszerzenie ts odpowiada plikom, które można bezpośrednio odczytać przy pomocy biblioteki **sktime**. Tak wczytane dane będą miały odpowiednią strukturę. Pozwoli to na użycie ich bezpośrednio w eksperymentach. Pliki ts dzielą się na dwie części: nagłówek oraz dane. Przykładowy plik może wyglądać następująco:

```
@problemName <problem name>
@timeStamps <true/false>
@univariate <true/false>
@classLabel <true/false> <oddzielone spacją wartości możliwych etykiet klas>
@data
2,3,2,4:4,3,2,2
13,12,32,12:22,23,12,32
4,4,5,4:3,2,3,2
```

Listing 4.1: Przykładowy początek pliku z rozszerzeniem .ts

Poniżej znacznika **data** wymienione są wartości kolejnych obserwacji w szeregu. Jeżeli występuje więcej niż jeden wymiar, oddziela się je przy pomocy dwukropka. Etykiety zawiera się w pliku dodając je na końcu każdego szeregu - również po dwukropku. Każdy szereg oddzielony jest nową linią. Powstałe w ten sposób zestawy danych dla obu badanych problemów to:

- Default - dane pochodzące z połączonego pliku Time_series_udf_dataset.csv. Posłużą do eksperymentów nieprzetworzonych danych.
- Normalized - dane pochodzące z połączonego pliku Time_series_udf_dataset_normalized.csv. Zastosuje się je do eksperymentów korzystających z znormalizowanych danych.
- Default smooth - dane pochodzące z połączonego pliku Time_series_udf_dataset_smooth_6.csv. Przeznaczone będą do eksperymentów używających wygładzonych surowych danych.
- Normalized smooth - dane pochodzące z połączonego pliku Time_series_udf_dataset_normali--zed_smooth_6.csv. Znajdą zastosowanie w eksperymentach z znormalizowanymi danymi, które dodatkowo zostały wygładzone.

Dzięki wcześniejszemu utworzeniu zbiorów w sposób podany powyżej, eksperymenty związane z uczeniem maszynowym oraz badaniem podobieństwa będą mogły zostać przeprowadzone na tych samych zestawach danych.

Rozdział 5

Eksperymenty

W tym rozdziale opisane zostały eksperymenty badające podobieństwo między przebiegami zużycia zasobów CPU oraz RAM podczas działania aplikacji przetwarzającej dane na klastrze. Wykonana została ich klasyfikacja algorytmami uczenia maszynowego oraz zastosowano algorytmy wyspecjalizowane w badaniu podobieństwa między szeregami czasowymi. Wykorzystane metody są opisane w rozdziale 2., a ich implementacje pochodzą z bibliotek wybranych w rozdziale 4.1. Eksperymenty zostały podzielone na dwa typy:

- **podobieństwo typów funkcji** - czy funkcje wykonujące ten sam rodzaj operacji są do siebie podobne
- **podobieństwo wolumenów danych** - czy funkcje wykonujące te same operacje są do siebie podobne, jeżeli rozmiar danych jest różny

Chcąc sprawdzić w pierwszym przypadku, czy typy przetwarzania są rozróżnialne, wykorzystano przykłady szeregów z różnych typów UDF. W ten sposób powstało pięć klas: *aggregation, filtering, aggregation-filtering, aggregation-filtering-join, filtering-join.* **Do testowania podobieństwa typów funkcji użyto razem danych zarówno o rozmiarze 1GB, jak i 2GB**. Drugi rodzaj eksperymentów sprawdza czy przetwarzanie w ramach typu operacji zachowuję podobieństwo, niezależnie od rozmiaru danych. W zbiorze badanym w ramach niniejszej pracy dostępne są dwa wolumeny danych, tworząc dwie klasy: 1GB i 2GB.

5.1 Podobieństwo typów funkcji

5.1.1 Algorytmy podobieństwa szeregów czasowych

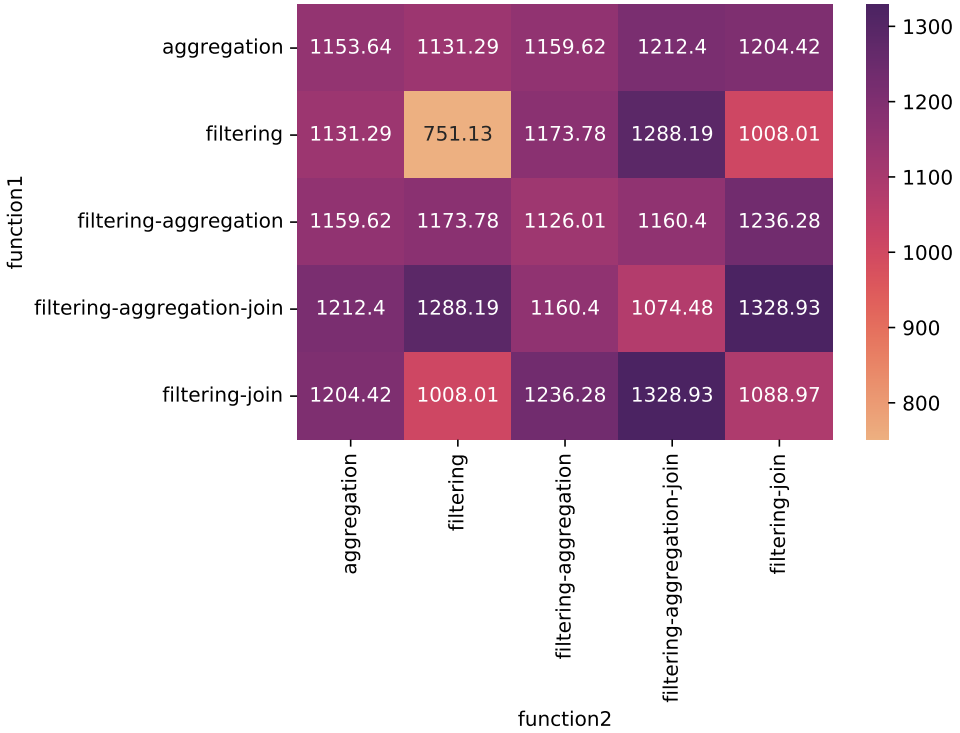
W eksperymentach z wykorzystaniem algorytmów podobieństwa użyto zbioru danych testowych w czterech wersjach: nieprzetworzonej, znormalizowanej (normalizacja opisana w sekcji 4.2.3), wygładzonej (wygładzanie opisane w sekcji 4.2.4) oraz znormalizowanej i wygładzonej. Każdy szereg czasowy połączono ze wszystkimi pozostałymi. Dla każdego takiego połączenia wyliczono odległości i podobieństwo wykorzystując biblioteki opisane w rozdziale 5.1. Okazało się, że biblioteka dtaidistance, implementująca Dynamic Time Warping, potrzebowała 10-krotnie więcej czasu na wykonanie obliczeń niż biblioteka dtw-python. Z tego względu do DTW wykorzystano dtw-python. Obliczenia wykonano w notatniku *8.1_calculate_distance_experiments.ipynb*.

Wyniki dla poszczególnych algorytmów

Porównano wyliczone odległości każdego typu funkcji z każdym. Wyniki przedstawiono za pomocą tablicy średnich odległości oraz wykresów gęstości. Wykres gęstości przedstawia rozkład odległości między szeregami czasowymi jednego z typów funkcji, a szeregami każdego typu. Wszystkie wykresy w tym podrozdziale zostały wygenerowane za pomocą notatnika *8.2_distance_analysis.ipynb*

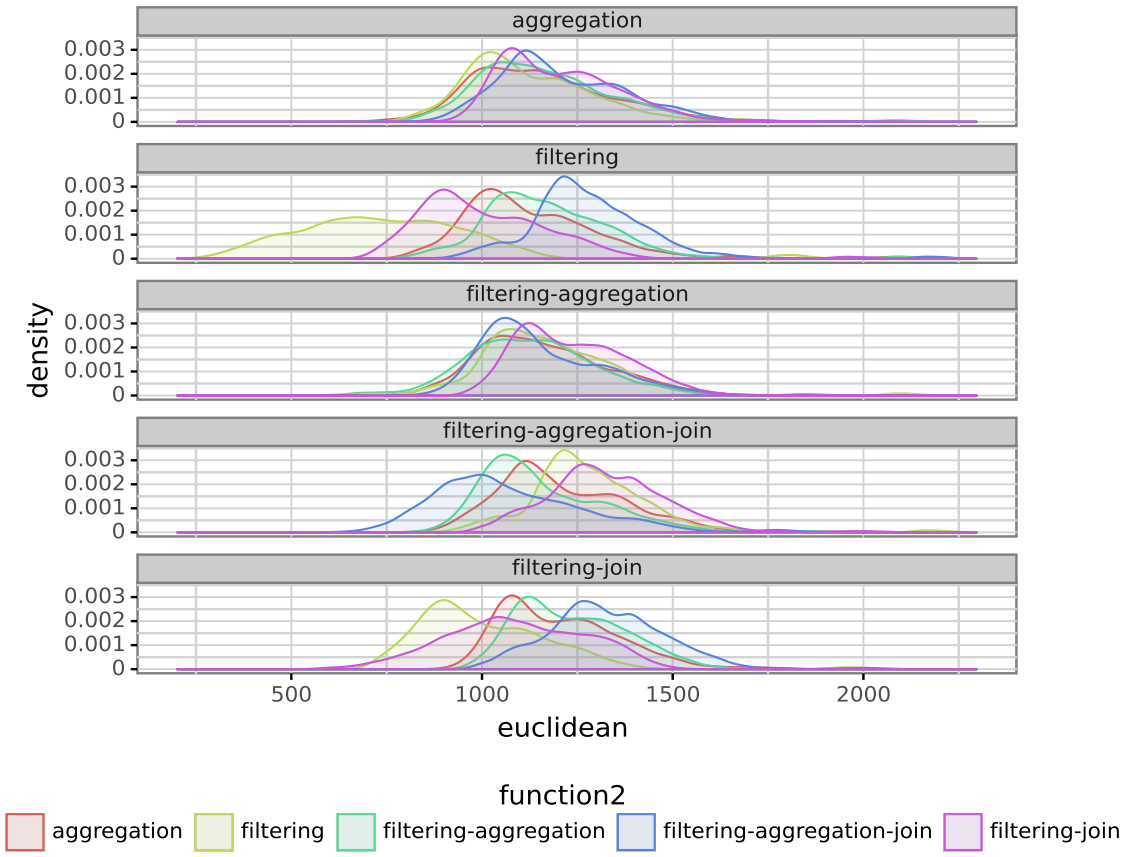
Odległość Euklidesowa

Rysunek 5.1 przedstawia średnie odległości uzyskane dla najprostszego podejścia - odległości euklidesowej dla nieprzetworzonych danych CPU. Nie pozwoliło ono na uzyskanie najlepszych wyników - można jednak zauważyć, że porównanie filtracji z filtracją daje najmniejszą wartość średnią. Wśród średnich z porównywania filtracji-agregacji-join i filtracji-agregacji najmniejszą wartość daje również porównanie z tym samym typem - przy filtracji-agregacji jest to jednak bardzo mała różnica. Dla filtracji-join sytuacja ma się inaczej - najmniejszą wartość średnią odległości daje porównanie z filtracją, dopiero na drugim miejscu jest filtracja-join, pozostałe typy są dużo bardziej odległe. Agregacja natomiast ma podobne wyniki dla każdego z typów.



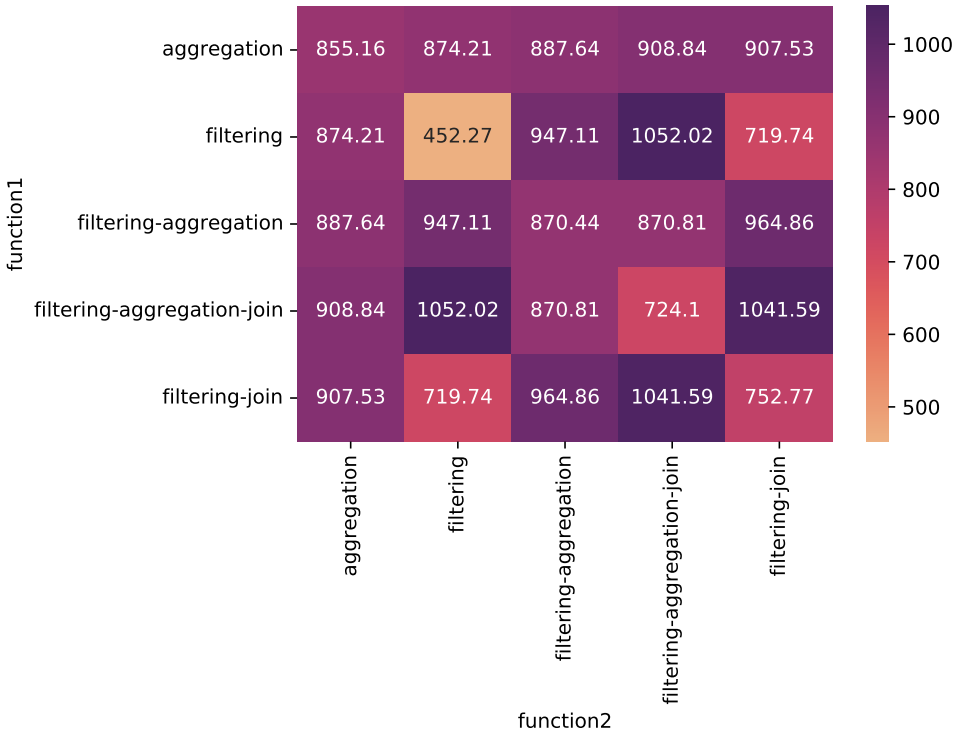
Rysunek 5.1: Heatmap - średnia odległość euklidesowa CPU dla danych nieprzetworzonych

Rysunek 5.2 przedstawia wykres gęstości odległości euklidesowej dla nieprzetworzonych danych. Potwierdza on wnioski wyciągnięte na podstawie wartości średnich.



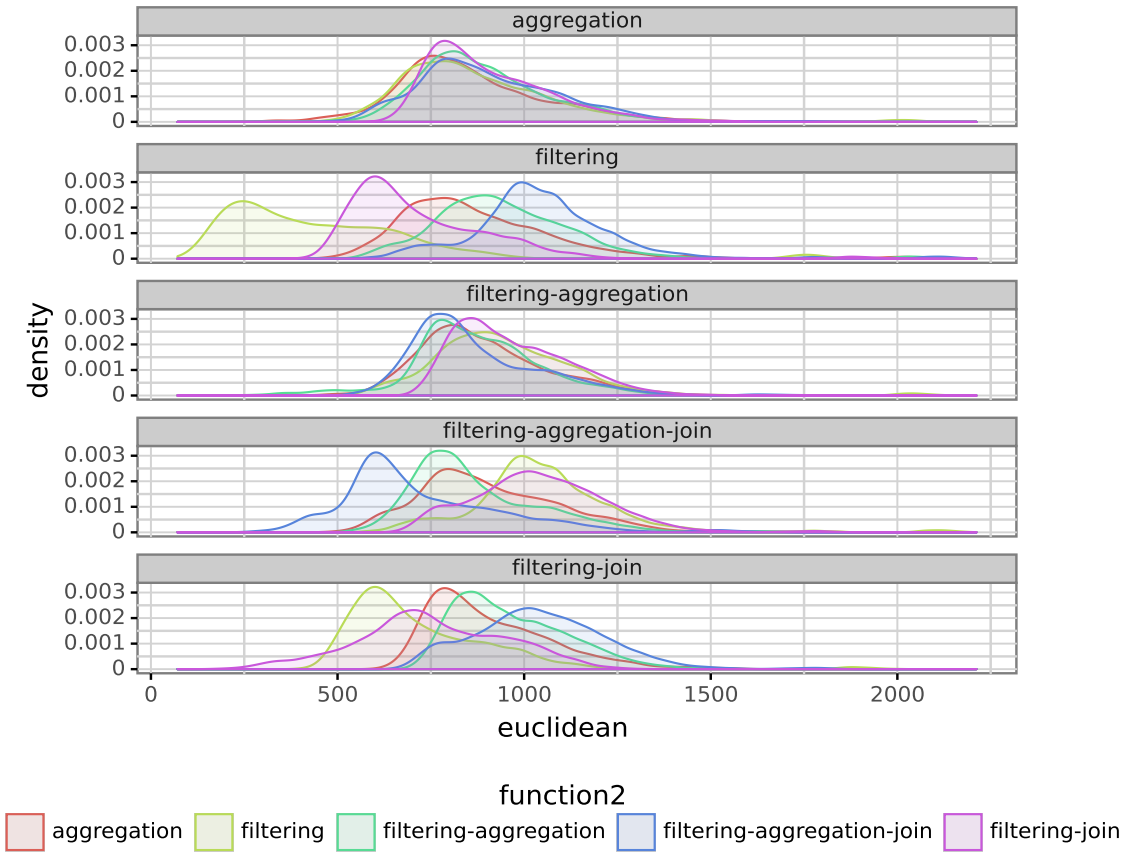
Rysunek 5.2: Wykres gęstości - Odległość euklidesowa CPU dla danych nieprzetworzonych

Średnie odległości euklidesowe dla danych wygładzonych CPU (rysunek 5.3) pozwalają na wysnucie podobnych wniosków, co dla danych nieprzetworzonych. Skutkiem wygładzenia jest to, że wszystkie odległości są nieco mniejsze. Pozytywnym efektem jest to, że wartość średnia między tymi samymi typami dla typu filtering-aggregation-join, jak i dla filtering-join zmniejszyła się w stosunku do pozostałych wartości.



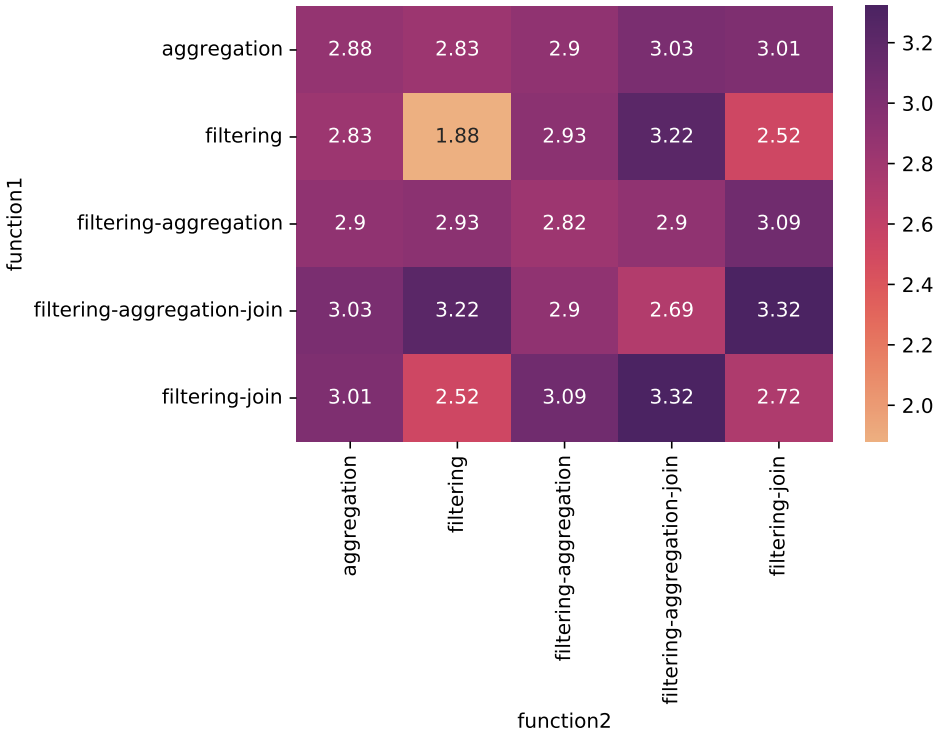
Rysunek 5.3: Heatmap - średnia odległość euklidesowa CPU dla danych wygładzonych

Na rysunku 5.4 można zauważyć, że po wygładzeniu danych, porównania przyjmujące jako jeden z szeregów, szereg typu filtracji i filtracji-agregacji-joina dają większą szansę na rozróżnienie. Wyraźnie widać, że średnio do filtracji najbardziej podobna jest inna filtracja, a następnie filtracja-join, agregacja, filtracja-agregacja, a na końcu filtracja-agregacja-join.

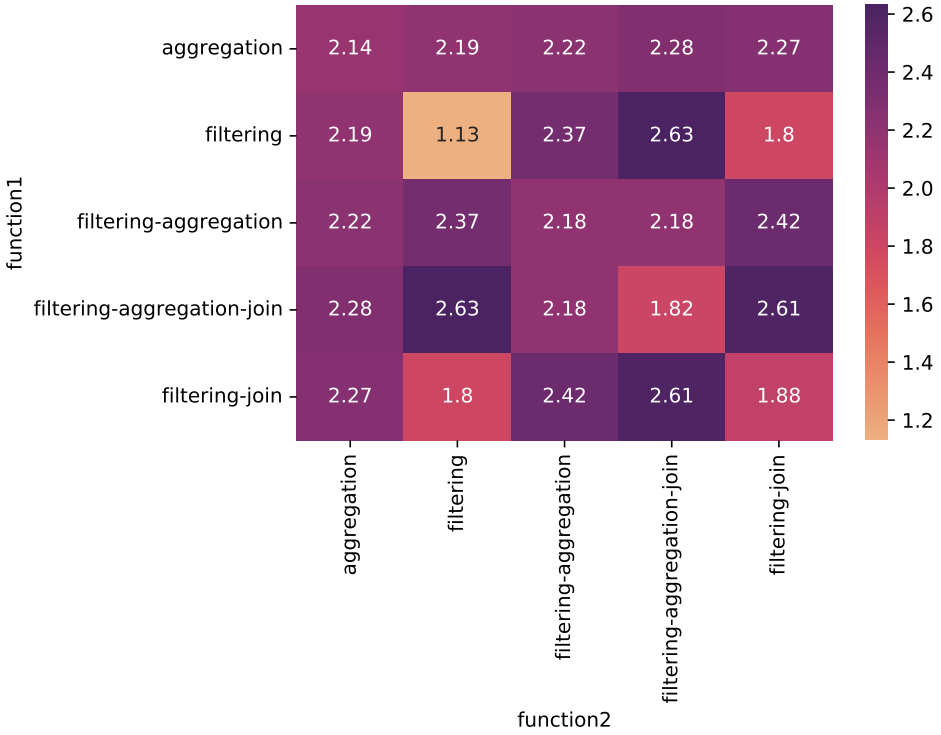


Rysunek 5.4: Wykres gęstości - Odległość euklidesowa CPU dla danych wygładzonych

Rysunek 5.5 przedstawia średnie dla znormalizowanych danych CPU. Na rysunku 5.6 można zobaczyć średnie dla znormalizowanych i jednocześnie wygładzonych danych CPU. Poza przeskalowanymi wartościami w wyniku normalizacji nie znaleziono w nich nowych informacji.

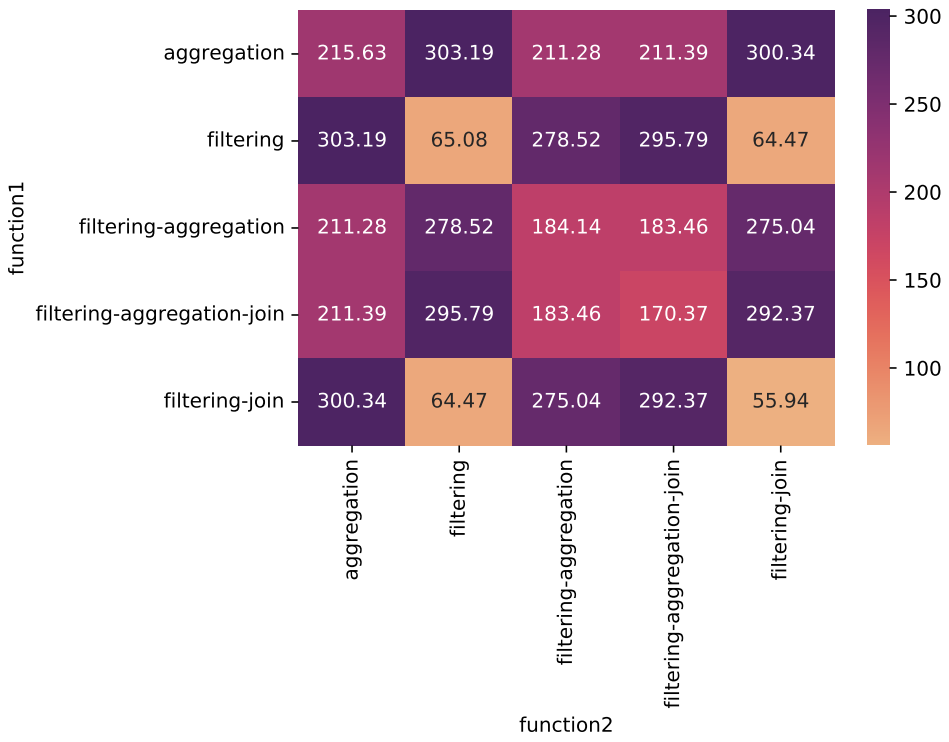


Rysunek 5.5: Heatmap - średnia odległość euklidesowa CPU dla danych znormalizowanych

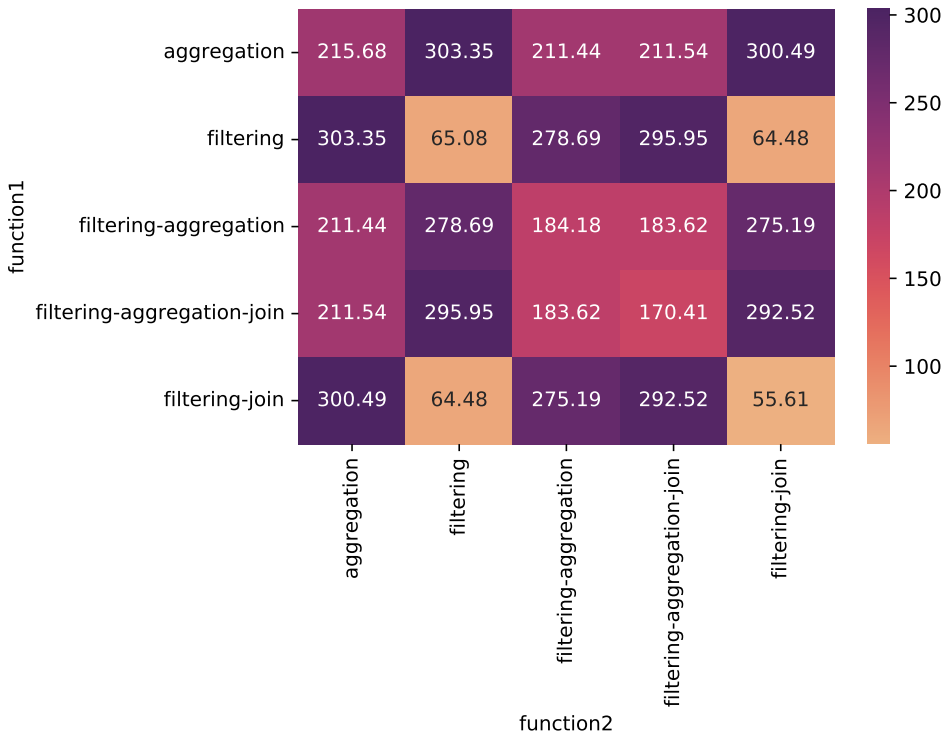


Rysunek 5.6: Heatmap - średnia odległość euklidesowa CPU dla danych znormalizowanych i wygładzonych

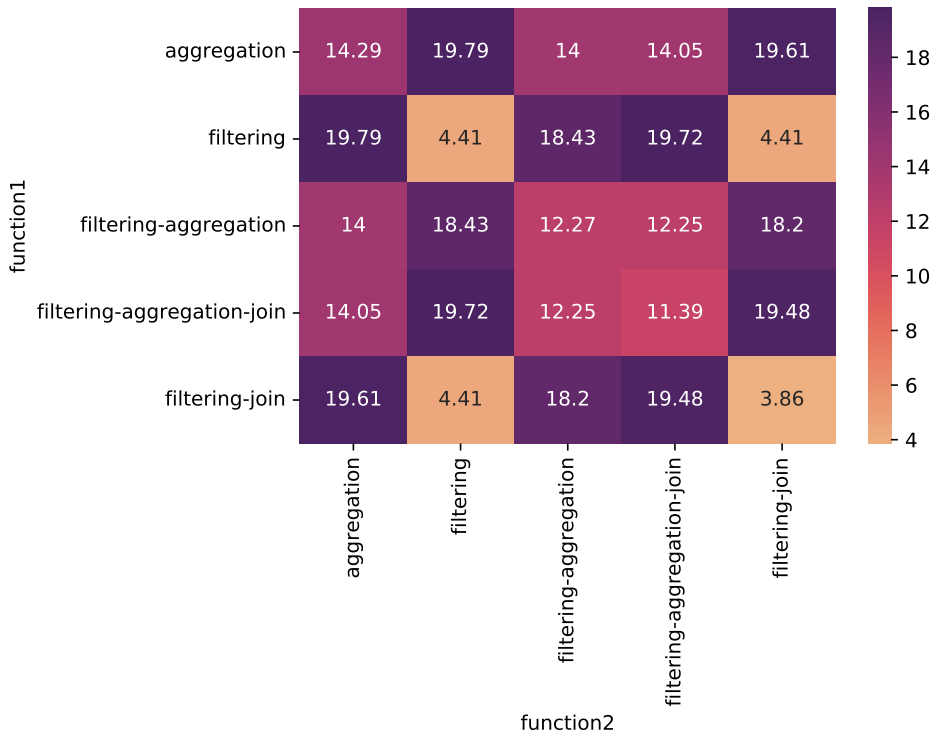
Rysunki 5.7, 5.8, 5.9, 5.10 przedstawiają wartości średnie odległości euklidesowej dla zużycia RAM kolejno dla danych nieprzetworzonych, wygładzonych, znormalizowanych oraz znormalizowanych i wygładzonych. Dane pozwalają jedynie stwierdzić wyraźne rozróżnienie zużycia pamięci RAM dla typów filtering i filtering-join, które są bardzo podobne, a pozostałymi typami.



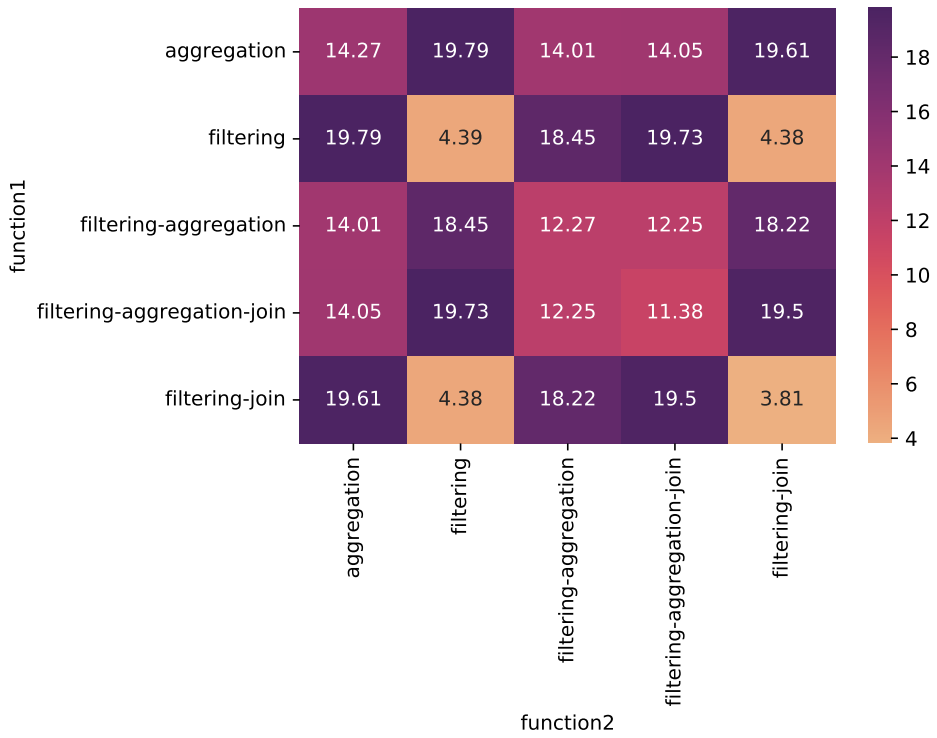
Rysunek 5.7: Heatmap - średnia odległość euklidesowa RAM dla danych nieprzetworzonych



Rysunek 5.8: Heatmap - średnia odległość euklidesowa RAM dla danych wygładzonych



Rysunek 5.9: Heatmap - średnia odległość euklidesowa RAM dla danych znormalizowanych

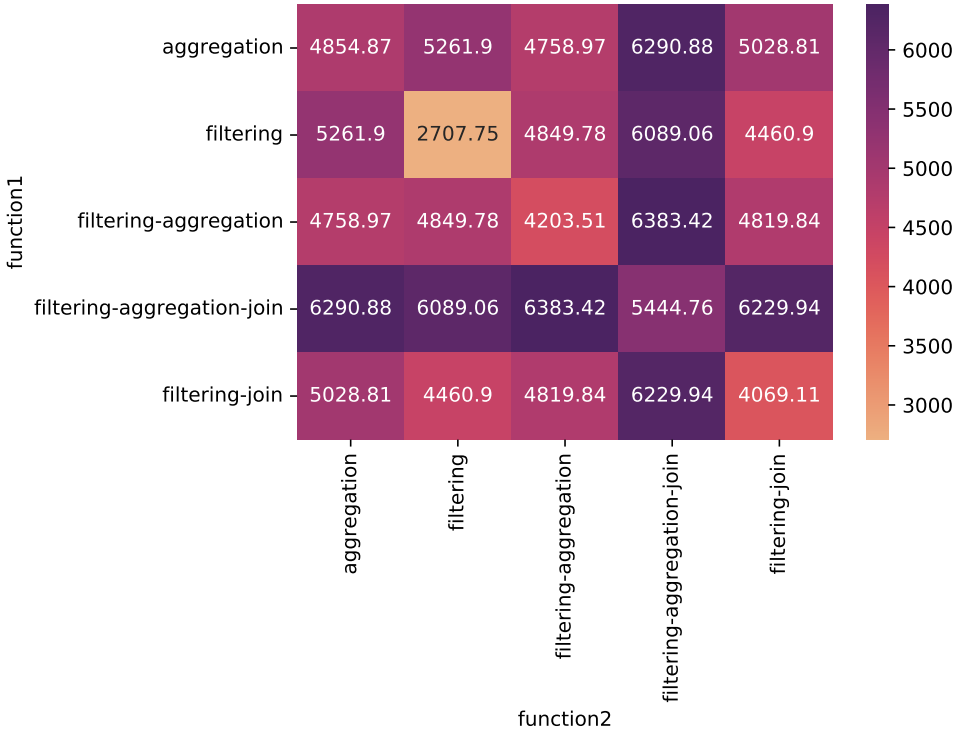


Rysunek 5.10: Heatmap - średnia odległość euklidesowa RAM dla danych znormalizowanych i wygładzonych

Dynamic Time Warping

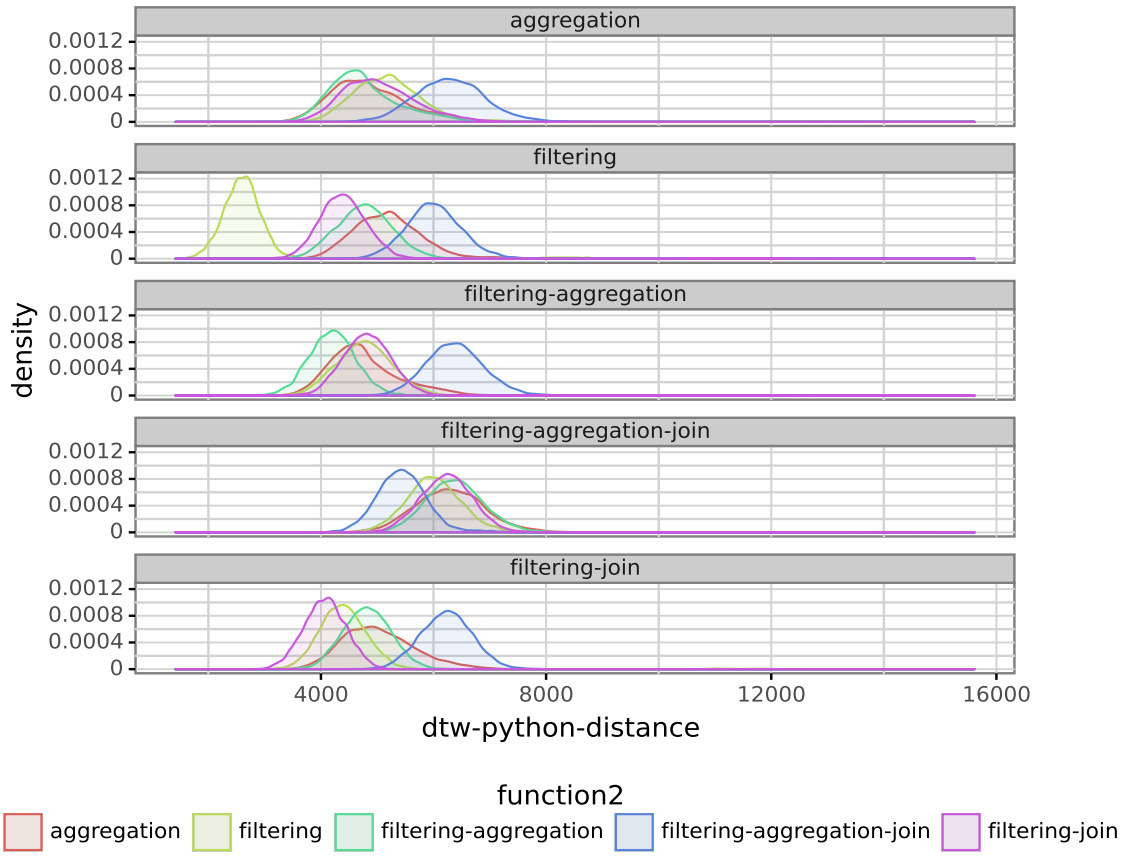
Rysunek 5.11 przedstawia tablicę średniej odległości między poszczególnymi typami dla danych nieprzetworzonych. Dla prawie każdego typu funkcji (poza agregacją), najbliższym typem funkcji

jest ten sam typ, jednak tak jak w przypadku odległości euklidesowej najbardziej wyróżnia się porównanie filtracji z filtracją, której wartość średnia jest dużo mniejsza od pozostałych. Funkcje typu filtering-aggregation-join są najbardziej różne od wszystkich pozostałych.

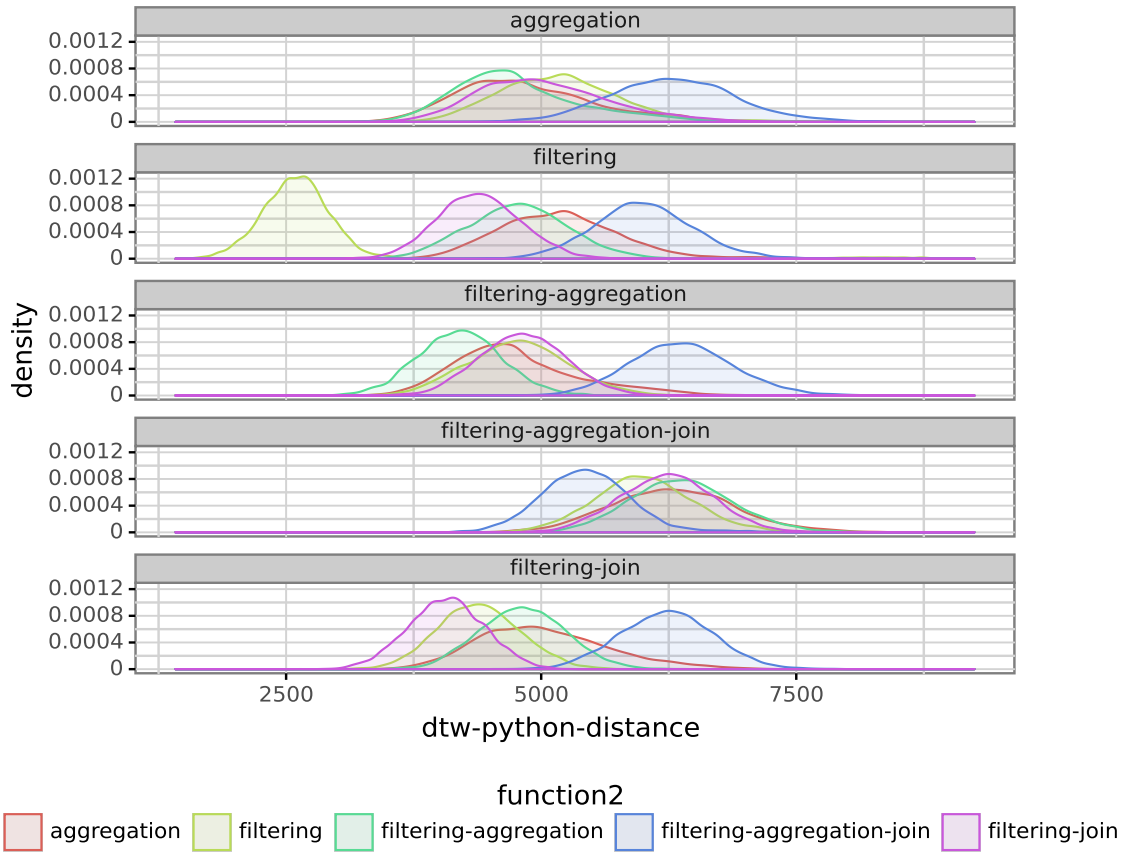


Rysunek 5.11: Heatmap - średnia odległość DTW CPU dla danych nieprzetworzonych

Rysunek 5.12 przedstawia wykres gęstości dla odległości Dynamic Time Warping zużycia CPU dla danych nieprzetworzonych. Można zauważyć, że wykresy mają kształt krzywej Gaussa. Część pomiarów pokrywa się do pewnego stopnia z innymi typami. Można zauważyć też, że pojawiło się sporo wartości odstających. Rysunek 5.13 pokazuje podobny wykres, usunięto jedynie odległości odstające, większe niż 10000.

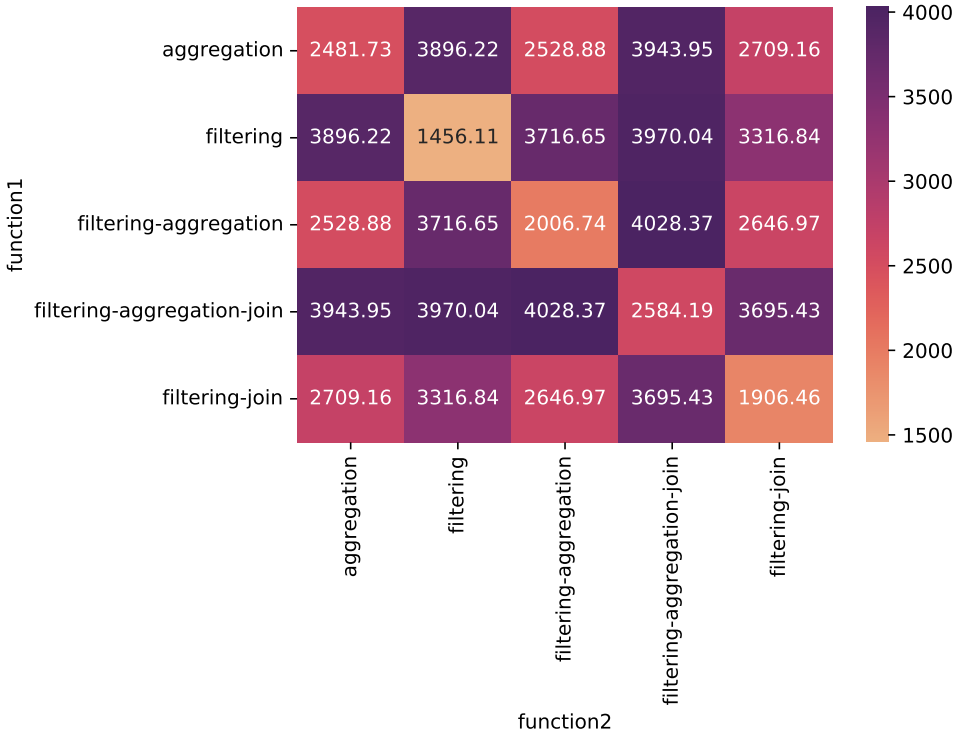


Rysunek 5.12: Wykres gęstości - odległość DTW CPU dla danych nieprzetworzonych



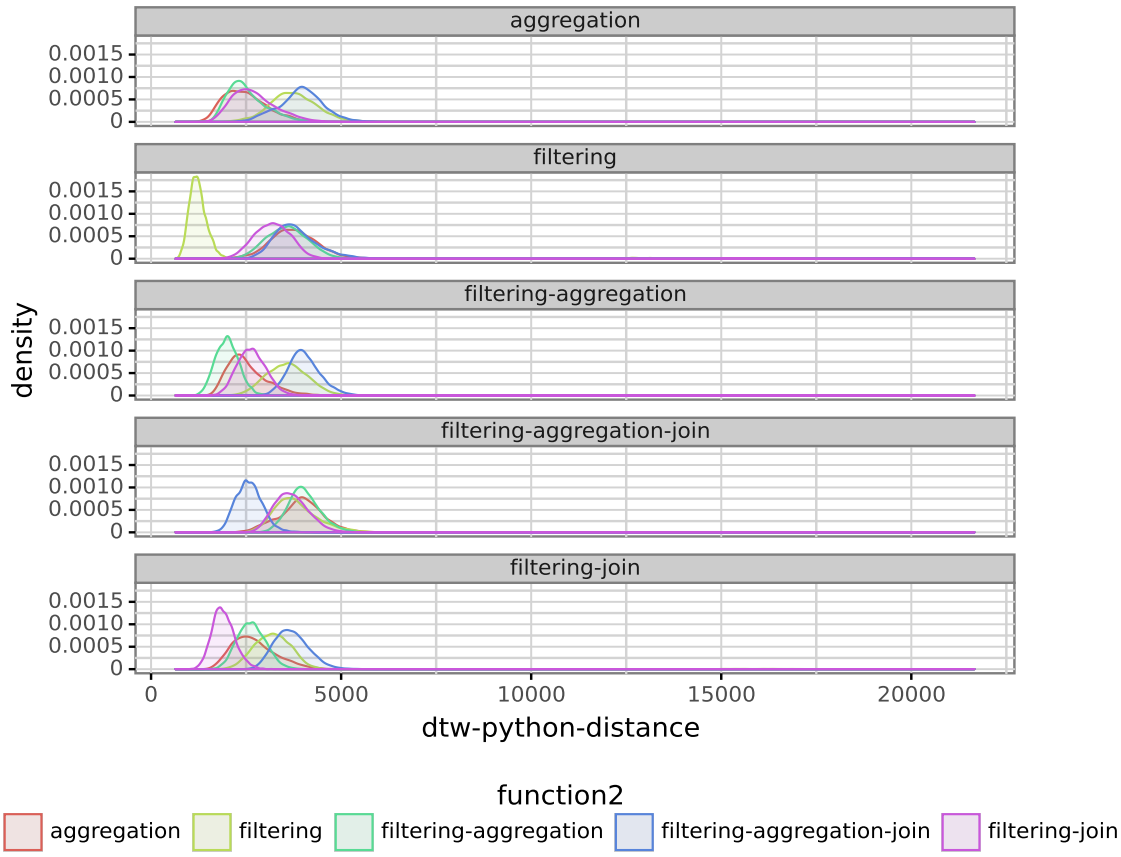
Rysunek 5.13: Wykres gęstości - odległość DTW CPU dla danych nieprzetworzonych - wartości mniejsze niż 10000

Rysunek 5.14 przedstawia tablicę średniej odległości między poszczególnymi typami dla wygładzonych danych zużycia CPU. W tym wypadku już każdy typ funkcji ma najmniejszą średnią odległość w porównaniu z szeregami tego samego typu. Różnice między porównaniami szeregów tego samego typu, a różnych typów wyraźnie się pogłębiły. Widać to zwłaszcza w stosunku do typu filtering-aggregation-join. Można powiedzieć, że wygładzenie danych miało w przypadku Dynamic Time Warping bardzo pozytywny wpływ na możliwość rozróżnienia typów funkcji.

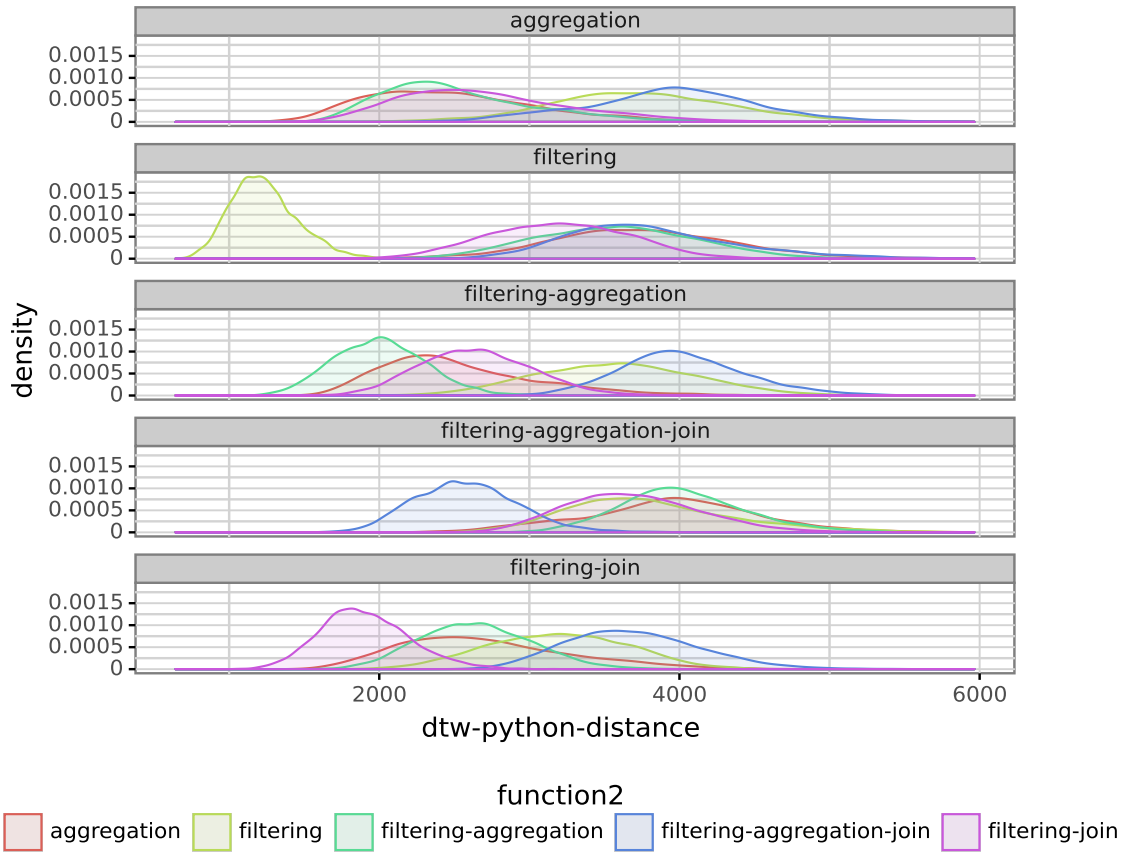


Rysunek 5.14: Heatmap - średnia odległość DTW CPU dla danych wygładzonych

Rysunek 5.15 przedstawia wykres gęstości dla odległości Dynamic Time Warping zużycia CPU dla danych wygładzonych. Najważniejszą nową obserwacją jest to, że znaczenie odległości odstających, mających większą wartość niż reszta znacząco się zwiększyło. Na rysunku 5.16 przedstawiony jest ten sam wykres, ale ograniczony do wartości mniejszych niż 6000.

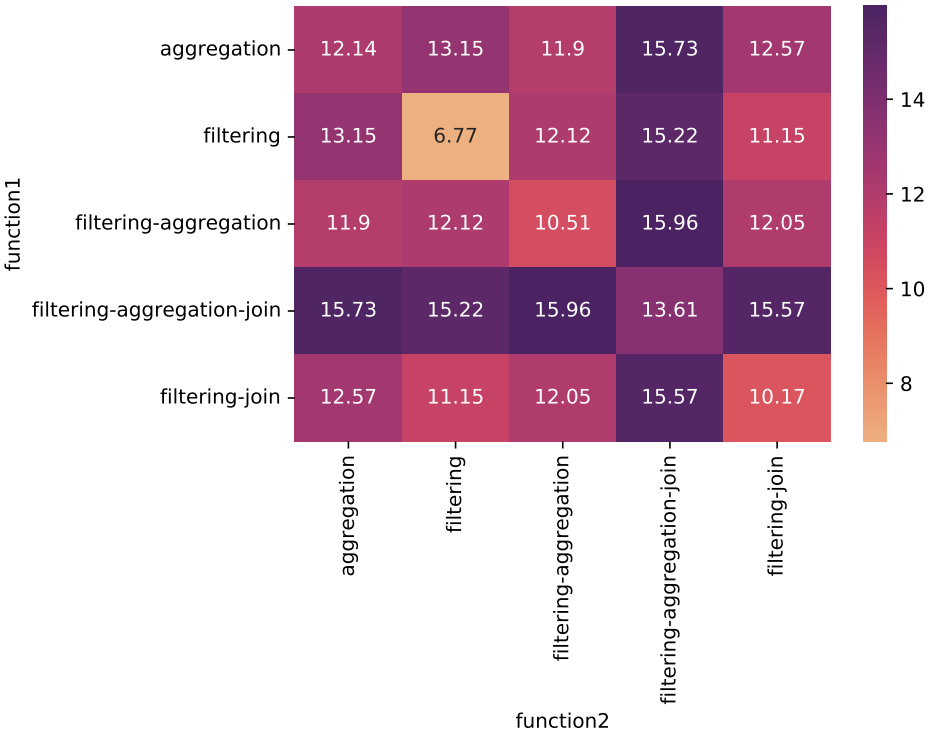


Rysunek 5.15: Wykres gęstości - odległość DTW CPU dla danych wygładzonych

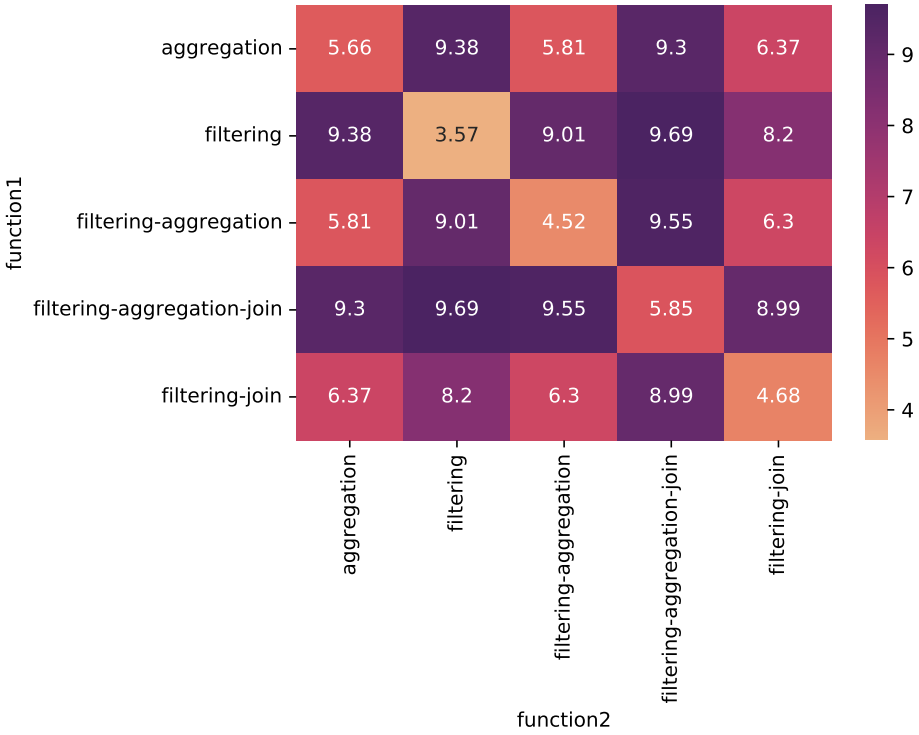


Rysunek 5.16: Wykres gęstości - odległość DTW CPU dla danych wygładzonych - wartości mniejsze niż 6000

Rysunek 5.17 przedstawia tablicę średnich odległości Dynamic Time Warping dla znormalizowanych danych zużycia CPU. Na rysunku 5.18 można zaobserwować tablicę średnich odległości dla znormalizowanych i jednocześnie wygładzonych danych CPU. Pierwsza tablica wygląda tak samo jak dla danych nieprzetworzonych, a druga tak samo jak dla danych wygładzonych, z tą różnicą, że wartości są w innej skali.



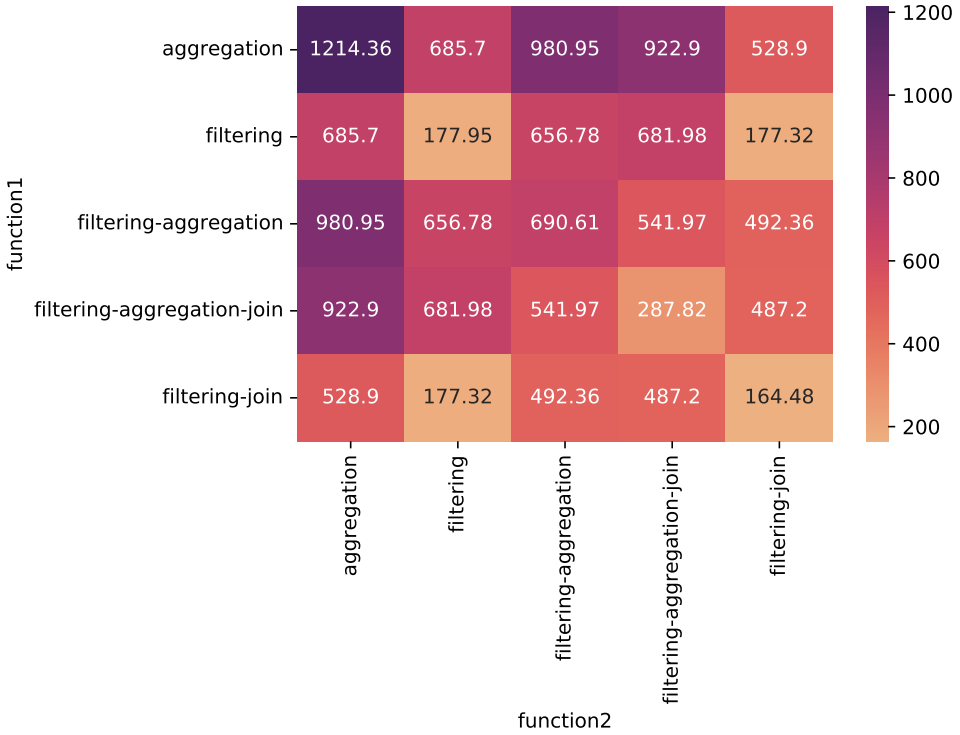
Rysunek 5.17: Heatmap - średnia odległość DTW CPU dla danych znormalizowanych



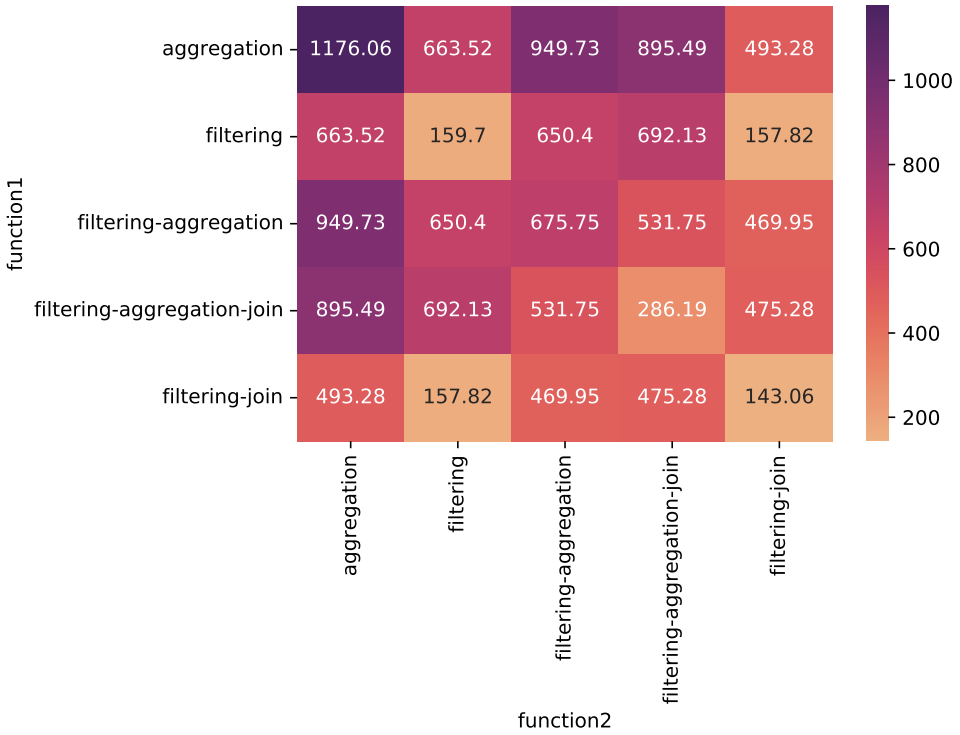
Rysunek 5.18: Heatmap - średnia odległość DTW CPU dla danych znormalizowanych i wygładzonych

Rysunki 5.19, 5.20, 5.21, 5.22 przedstawiają tablice średniej odległości Dynamic Time Warping zużycia RAM kolejno dla danych nieprzetworzonych, wygładzonych, znormalizowanych oraz znormalizowanych i wygładzonych. Podobnie jak w przypadku odległości euklidesowej dla danych zużycia RAM, można łatwo odróżnić filtrację i filtrację join od pozostałych. Również porówna-

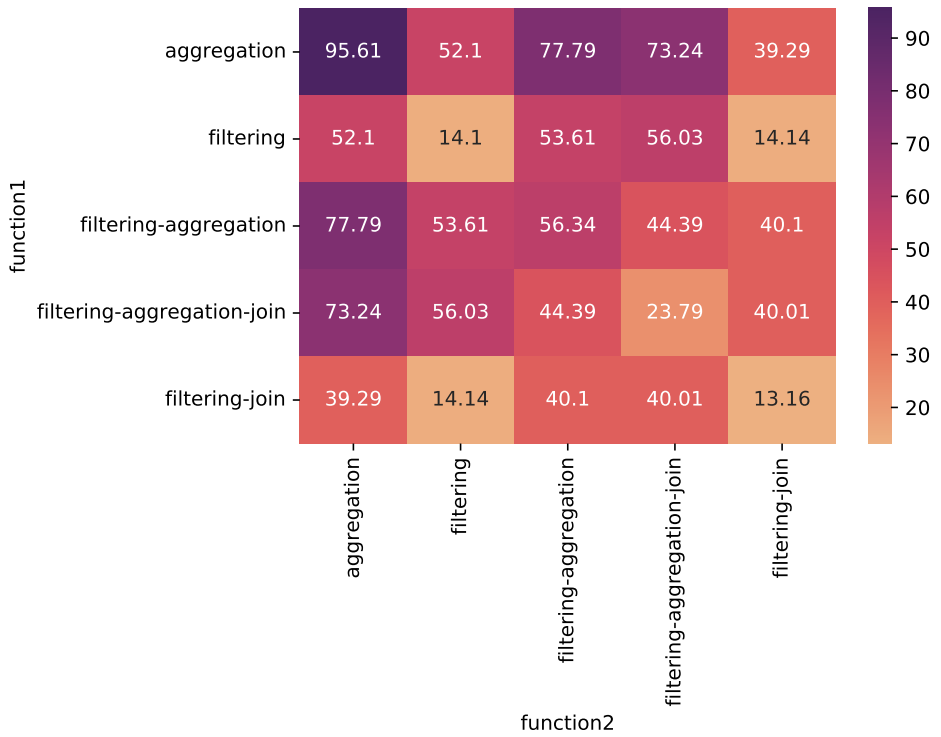
nie filtracji-agregacji-join ze sobą daje wyraźnie mniejszą odległość niż jej porównania z innymi funkcjami.



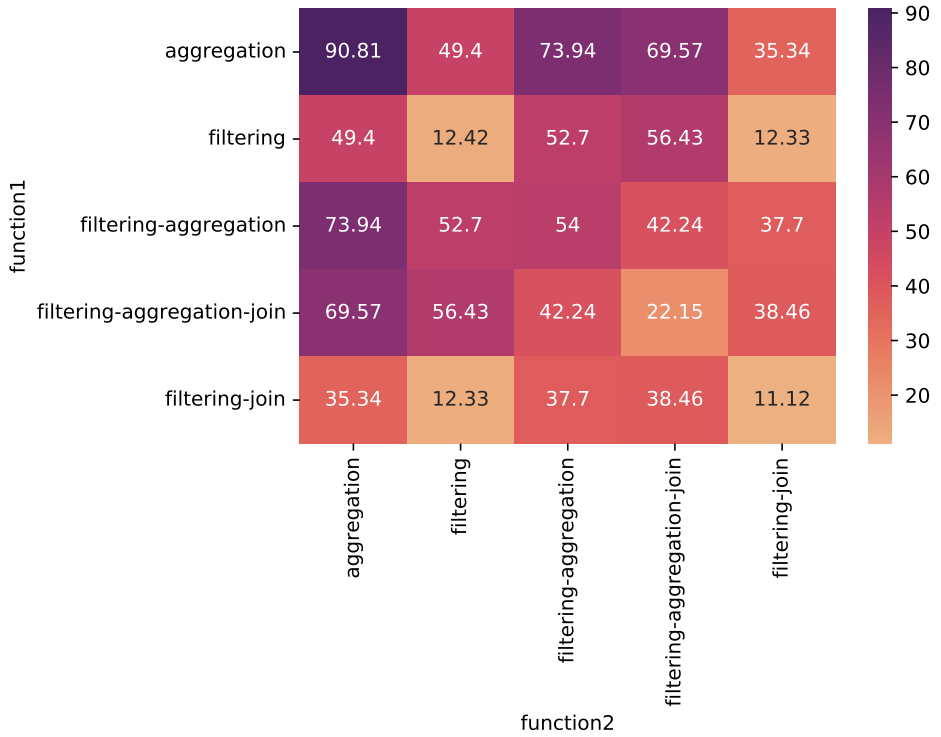
Rysunek 5.19: Heatmap - średnia odległość DTW RAM dla danych nieprzetworzonych



Rysunek 5.20: Heatmap - średnia odległość DTW RAM dla danych wygładzonych

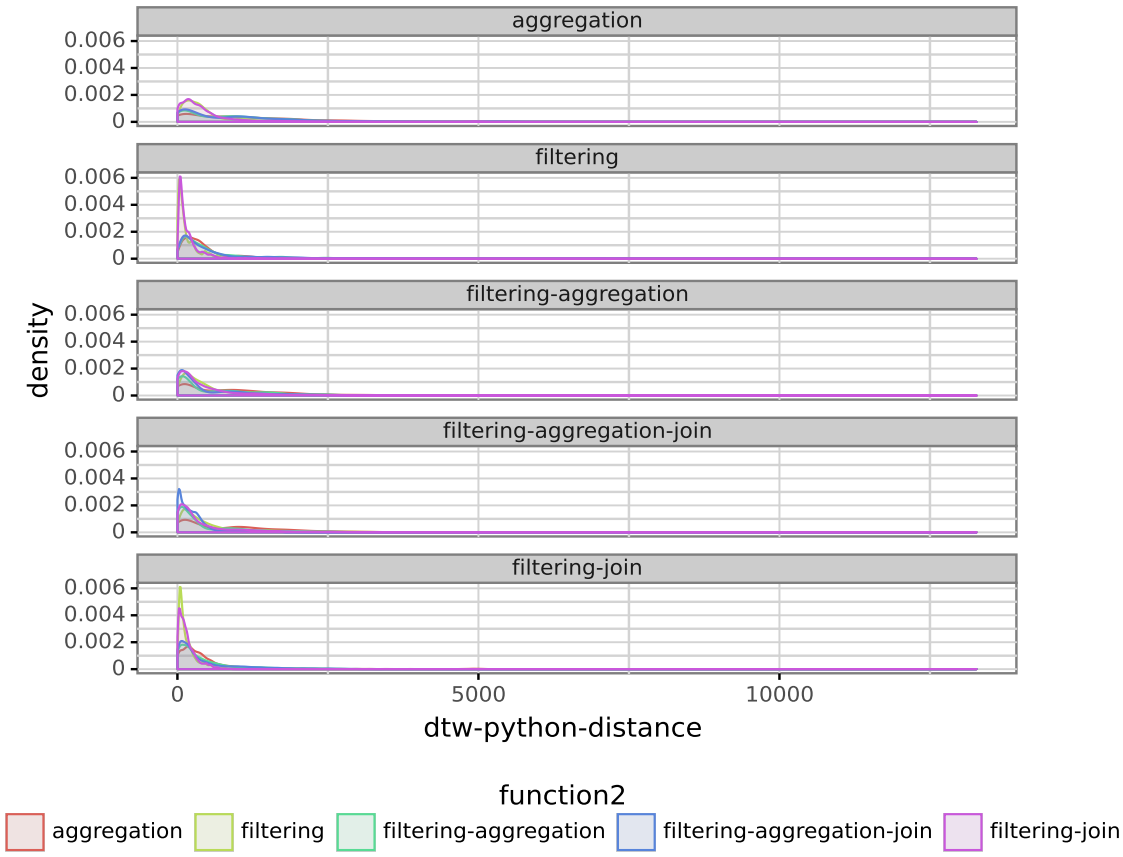


Rysunek 5.21: Heatmap - średnia odległość DTW RAM dla danych znormalizowanych



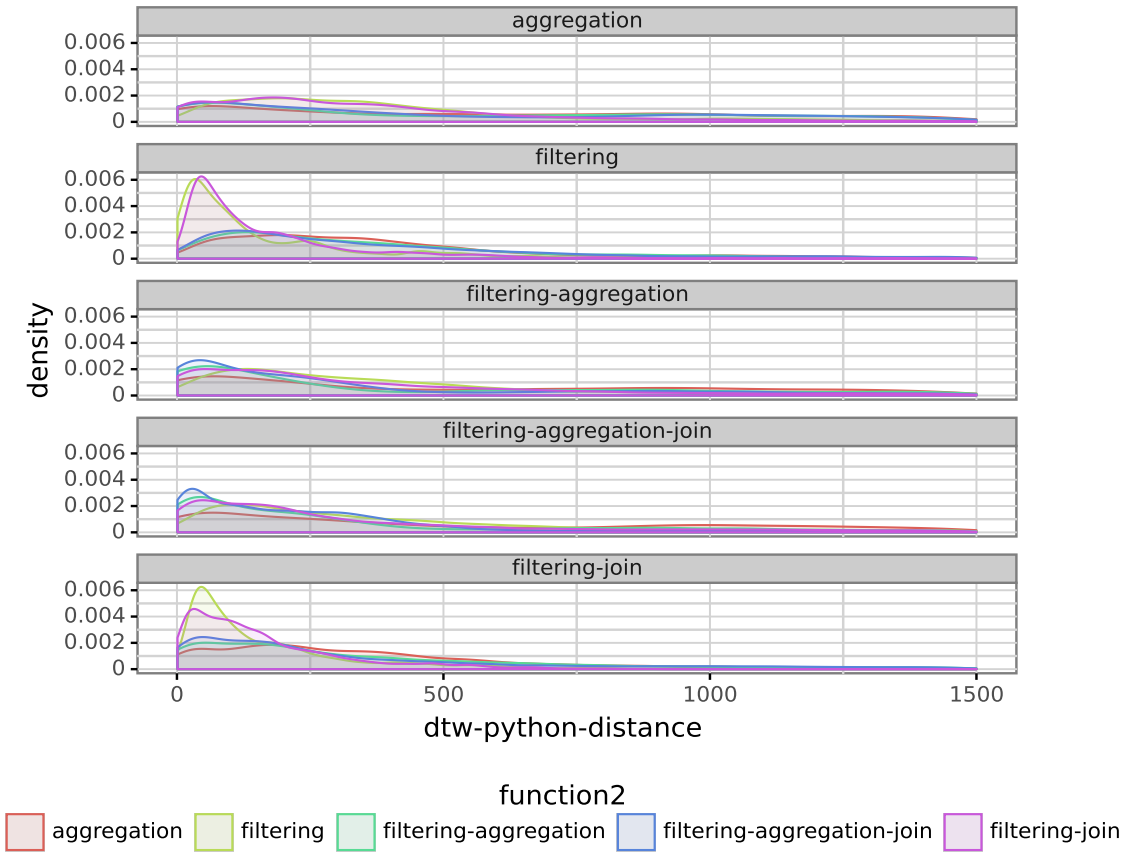
Rysunek 5.22: Heatmap - średnia odległość DTW RAM dla danych wygładzonych i znormalizowanych

Na rysunku 5.23 widać wykres gęstości dla odległości Dynamic Time Warping zużycia RAM dla danych nieprzetworzonych. Wynika z niego, że część odległości wyróżnia się na tle reszty. Obliczono, że odległość większą niż 1000 ma około 18% porównań, a większą niż 1500 ma 11%.



Rysunek 5.23: Wykres gęstości - odległość DTW RAM dla danych nieprzetworzonych

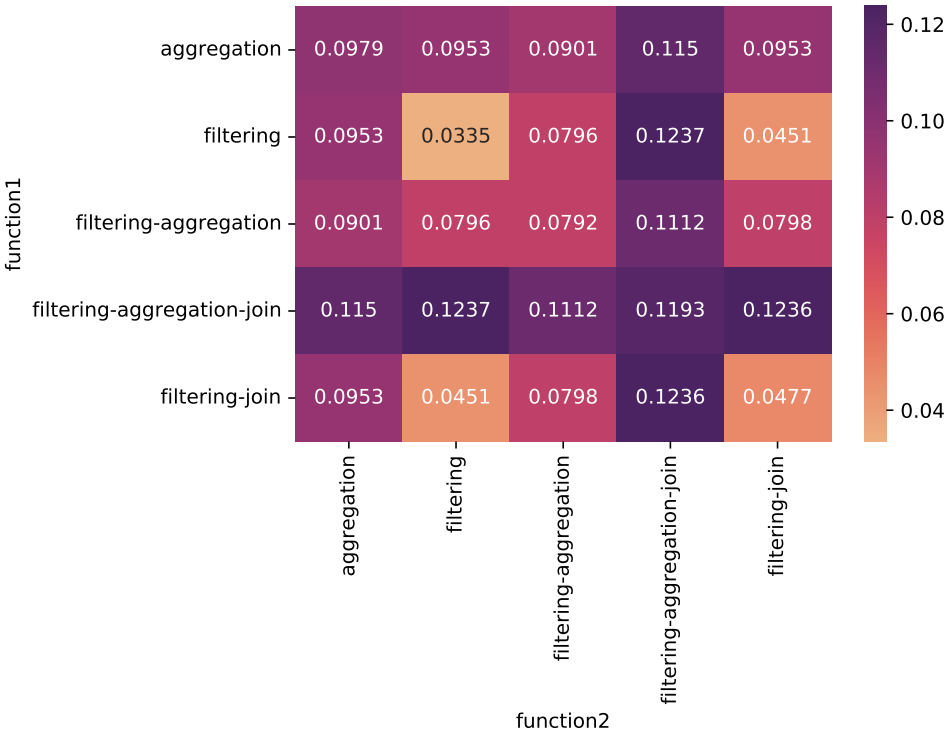
Rysunek 5.24 przedstawia wykres gęstości dla odległości Dynamic Time Warping zużycia RAM dla danych nieprzetworzonych ograniczony do wartości mniejszych niż 1500. Pozwala na zauważenie właściwości opisanych wcześniej. Filtracja i filtracja-join mają średnio wyraźnie mniejszą odległość niż pozostałe typy.



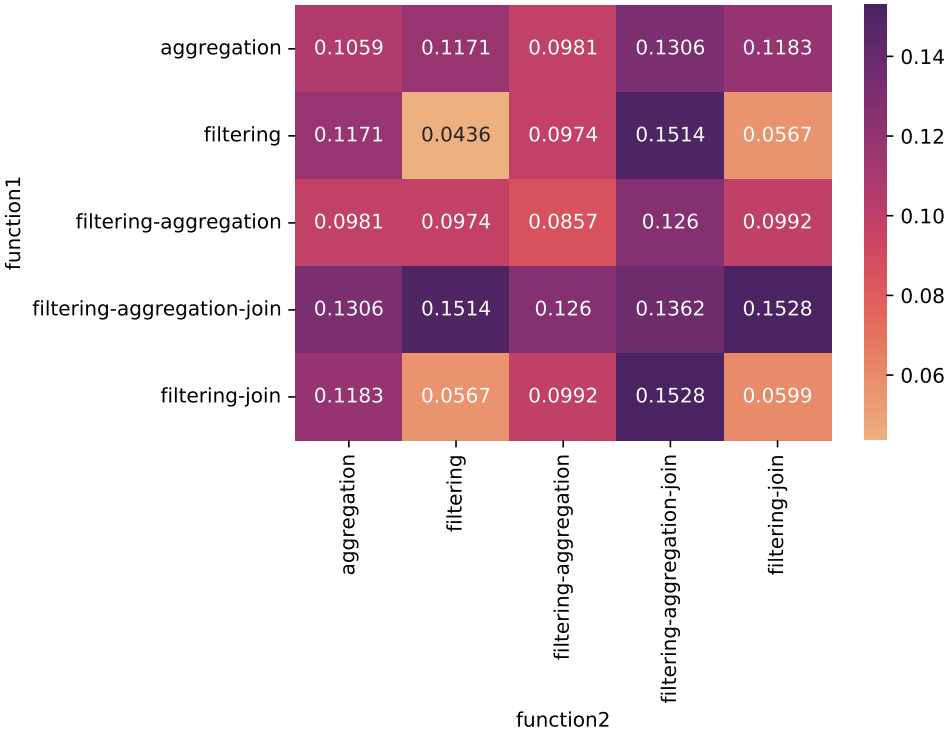
Rysunek 5.24: Wykres gęstości - odległość DTW RAM dla danych nieprzetworzonych, odległości mniejsze od 1500

Longest Common Subsequence

Żeby zachować spójność z resztą algorytmów, wynik algorytmu Longest Common Subsequence odjęto od 1. Konieczność tego działania była spowodowane przez fakt, że LCSS zwraca podobieństwo. W ten sposób otrzymano odległość. Rysunki 5.25, 5.26 przedstawiają tablice średnich odległości Longest Common Subsequence, dla zużycia CPU dla danych nieprzetworzonych i wygładzonych. Widać na nich wyraźnie, że filtracja i filtracja join odznaczają się na tle reszty. Można zauważyć też, że agregacja i filtracja-agregacja zachowują się podobnie. Warte odnotowania jest również to, że chociaż odległość może być liczbą z zakresu 0-1, to wszystkie wyliczone średnie odległości były mniejsze od 0.2. Jest to skutkiem tego, że przebiegi CPU składały się w dużej mierze z zer niezależnie od typu funkcji. Fragmenty te zostały uznane za podobne.



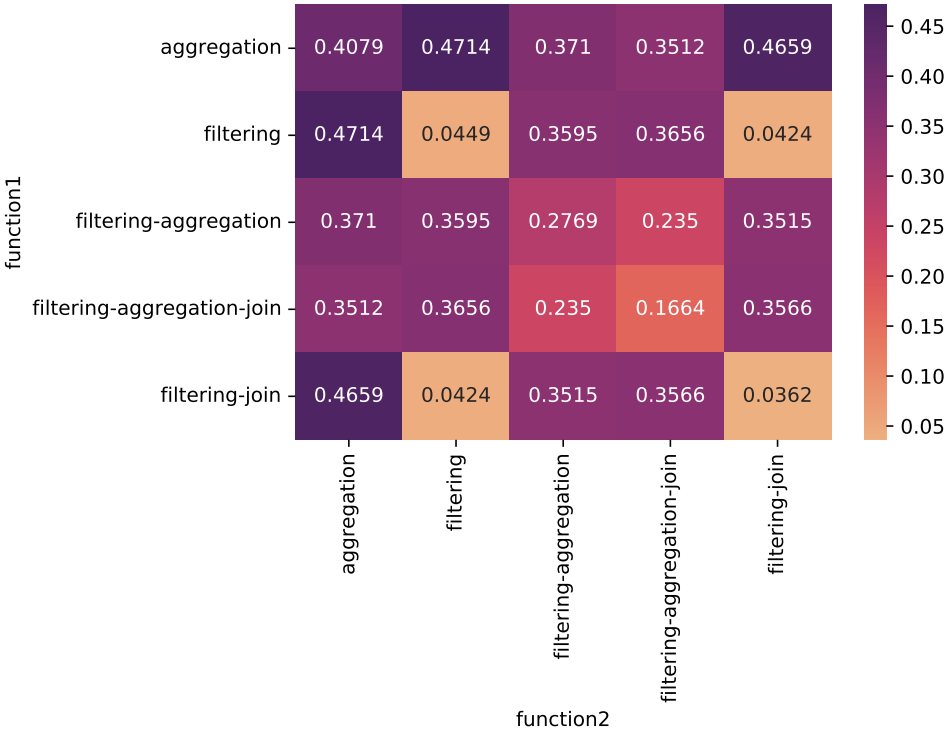
Rysunek 5.25: Heatmap - średnia odległość LCSS CPU dla danych nieprzetworzonych



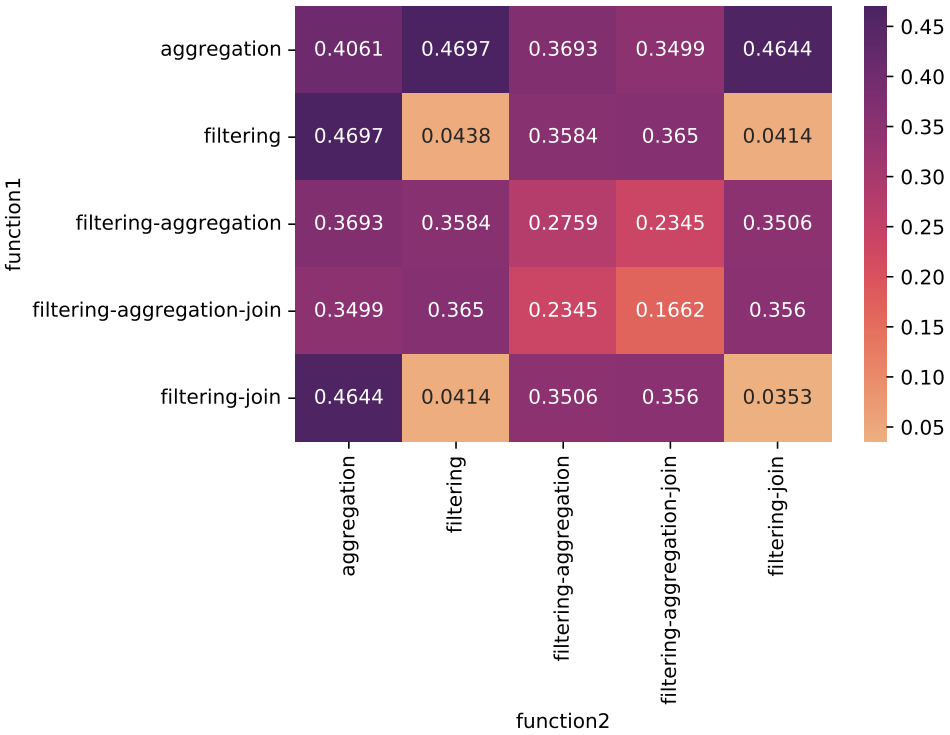
Rysunek 5.26: Heatmap - średnia odległość LCSS CPU dla danych wygładzonych

Rysunki 5.27, 5.28 przedstawiają tablice średnich odległości Longest Common Subsequence, dla zużycia CPU dla danych nieprzetworzonych i wygładzonych. Widać na nich przede wszystkim to, że filtracja i filtracja-join są bardzo do siebie podobne. Można zauważyć też, że odległość między filtracją-join, a filtracją-agregacją-join jest duża. Szeregi czasowe dla agregacji są w podobnej

odległości do wszystkich typów, nawet do pozostałych szeregów z agregacją.



Rysunek 5.27: Heatmap - średnia odległość LCSS RAM dla danych nieprzetworzonych



Rysunek 5.28: Heatmap - średnie odległość LCSS RAM dla danych wygładzonych

5.1.2 Algorytmy uczenia maszynowego

Wszystkie eksperymenty w niniejszej sekcji zostały przeprowadzone przy pomocy notatnika *9.1_run_machine_learning_type_experiments.ipynb*. Tworzy on pliki, które zawierają rezultaty klasyfikacji każdego z badanych algorytmów, zapisane w folderze *MachineLearning/results*. Wizualizacja niniejszych eksperymentów została wykonana przy pomocy notatnika *9.3_results_for_type_machine_learning_experiments.ipynb*. i wszystkie grafy zapisane zostały w folderze *Plots*.

Algorytmy HIVECOTEV2 oraz ROCKET wykorzystują generator liczb losowych podczas tworzenia modelu, np. losowane są parametry kerneli konwolucyjnych w algorytmie ROCKET. Z tego względu jednym z parametrów wejściowych jest liczba inicjująca te generatory. W celu upewnienia się, że dla rożnych wartości inicjujących (ang. *seed*) badane implementacje algorytmów zachowują się podobnie, każdy z eksperymentów został powtórzony dziesięć razy, z różnymi wartościami tego parametru. W przypadku algorytmu KNN przetwarzanie powtórzono dla trzech różnych wartości parametru k. Z względu na to, że jedyny sposób wyboru parametru k to wykonywanie eksperymentów, w niniejszej pracy wykorzystano wartości 1, 3 oraz 5. Są one często wykorzystywane w literaturze, a wartości nieparzyste zmniejszają prawdopodobieństwo remisu podczas klasyfikacji [9] [10]. Wyniki zostały uśrednione w ramach każdego z przygotowanych zbiorów, czyli *Default, Default Smooth, Normalized, Normalized Smooth* (zbiory te zostały opisane w sekcji 4.2.5). Służyło to znalezieniu zbioru, dla którego algorytmy uzyskują najwyższe wartości trafności (wzór 2.1) oraz F-miary (wzór 2.7). Każdy z badanych algorytmów miał możliwość wykorzystania wielowymiarowych szeregów czasowych. Z tego powodu **do klasyfikacji wykorzystane są przebiegi zużycia zarówno CPU jak i RAM.**

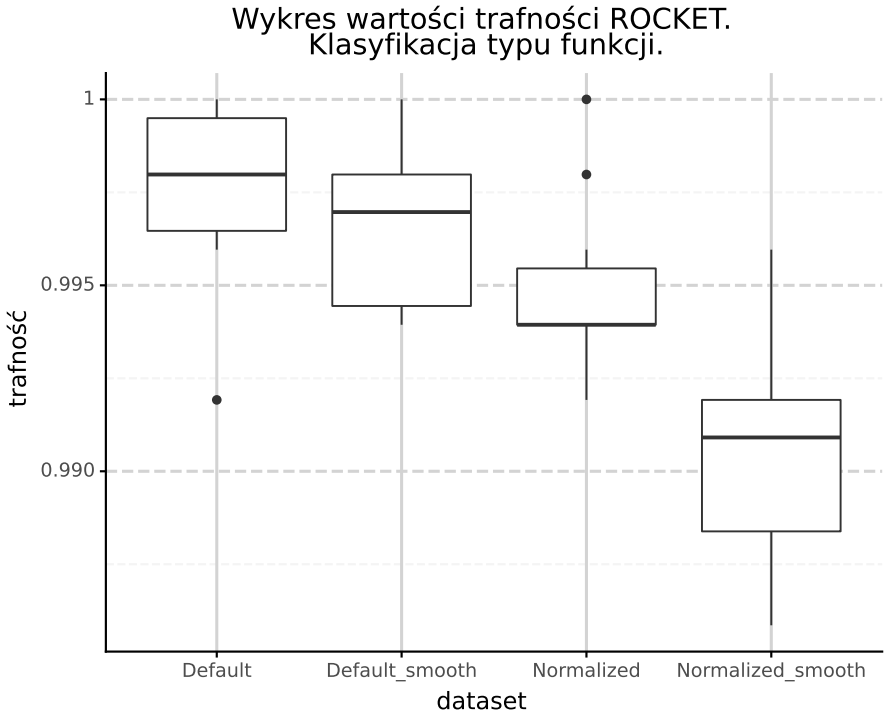
Wyniki metody ROCKET

Pierwszym sprawdzonym algorytmem był ROCKET, opisany w rozdziale 2. Jest to jeden z najnowocześniejszych algorytmów badających szeregi czasowe. Średnio najlepsze wyniki uzyskał on dla nieprzetworzonego zbioru danych, czyli *Default*. Trafność dla tego zestawu wyniosła aż 0.998. F-miara uzyskała taki sam wynik. Tablica 5.1 zawiera zebrane średnie wartości trafności i F-miary wraz z średnim czasem przetwarzania dla poszczególnych zbiorów danych.

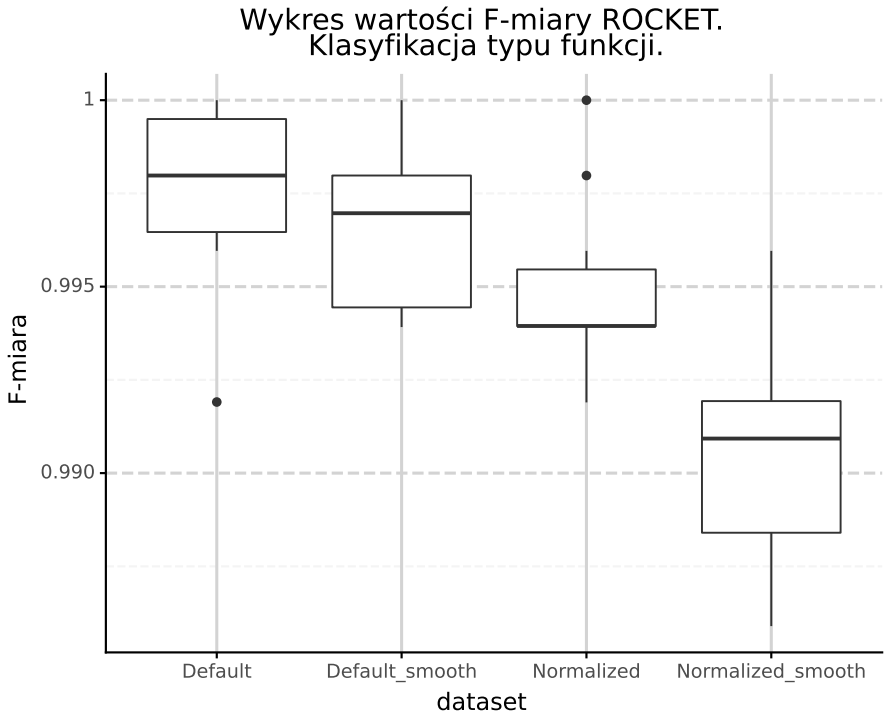
Tablica 5.1: Wyniki eksperymentów algorytmu ROCKET dla klasyfikacji typów funkcji

zestaw danych (dataset)	trafność	f-miara	czas przetwarzania (s)
Default	0.998	0.998	63.160
Default_smooth	0.997	0.997	62.731
Normalized	0.995	0.995	63.427
Normalized_smooth	0.991	0.991	67.108

Można zauważyć, że dla tego algorytmu normalizacja, jak i wygładzanie danych miały negatywny wpływ na wyniki. Czas przetwarzania algorytmu wyniósł średnio nieco ponad jedną minutę dla każdego powtórzenia. Rysunek 5.29 przedstawia wykres pudełkowy uzyskanych wyników. Można na nim zauważyć, że algorytm działa stabilnie, co objawia się niewielkim rozmiarem pudełek. Oznacza to, że czynnik losowy miał niewielki wpływ na uzyskane rezultaty.



(A) Trafność

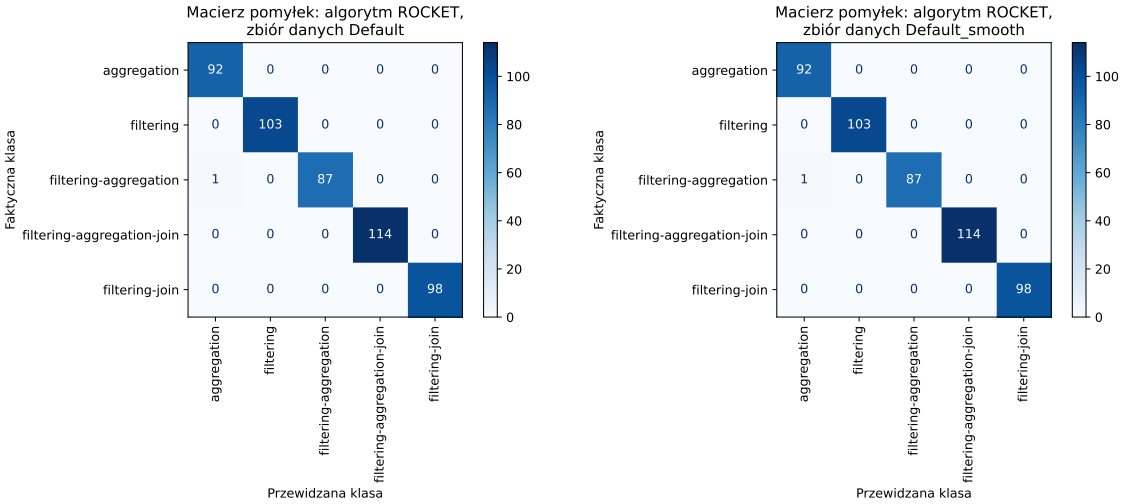


(B) F-miara

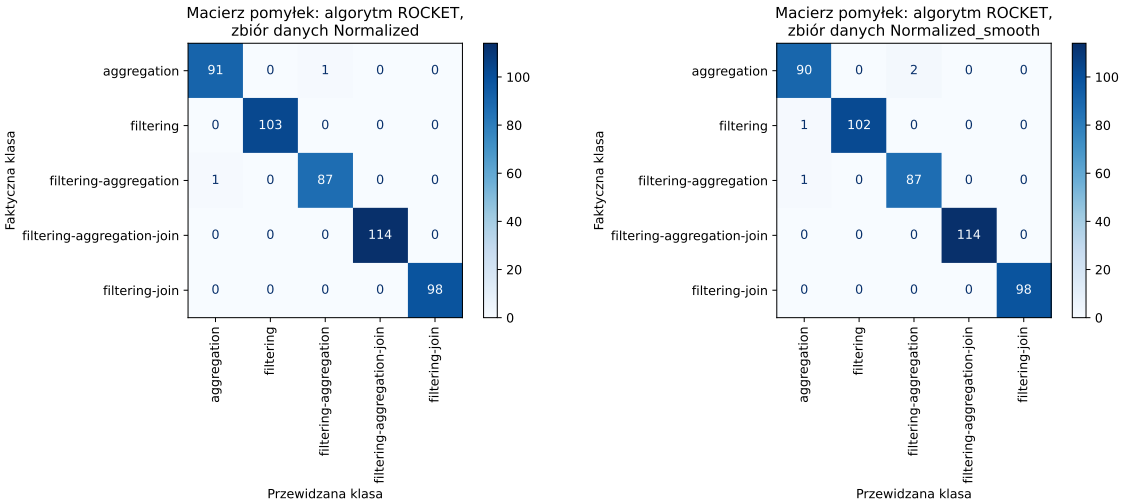
Rysunek 5.29: Wykresy pudełkowe trafności oraz F-miary dla algorytmu ROCKET.

Wyniki dla algorytmu ROCKET są zbliżone do 100% skuteczności klasyfikacji na zadanym zbiorze. Występują jednak pojedyncze przypadki błędnego przypisania etykiety. Rysunek 5.30 ukazuje macierze pomyłek dla każdego ze zbiorów danych. Należą one do eksperymentów, których trafność była najbardziej zbliżona do średniej trafności danego zestawu. Można na nich zauważyć, że jedyny problematyczny element stanowi klasa aggregation. Każdy błąd w macierzach pomy-

łek powiązany był z tym typem operacji. Są to jednak pojedyncze przypadki, które nie mają znaczącego wpływu na ogólne wyniki eksperymentu.



(A) Default (B) Default_smooth



(C) Normalized (D) Normalized_smooth

Rysunek 5.30: Macierze pomyłek dla każdego zbioru danych algorytmu ROCKET

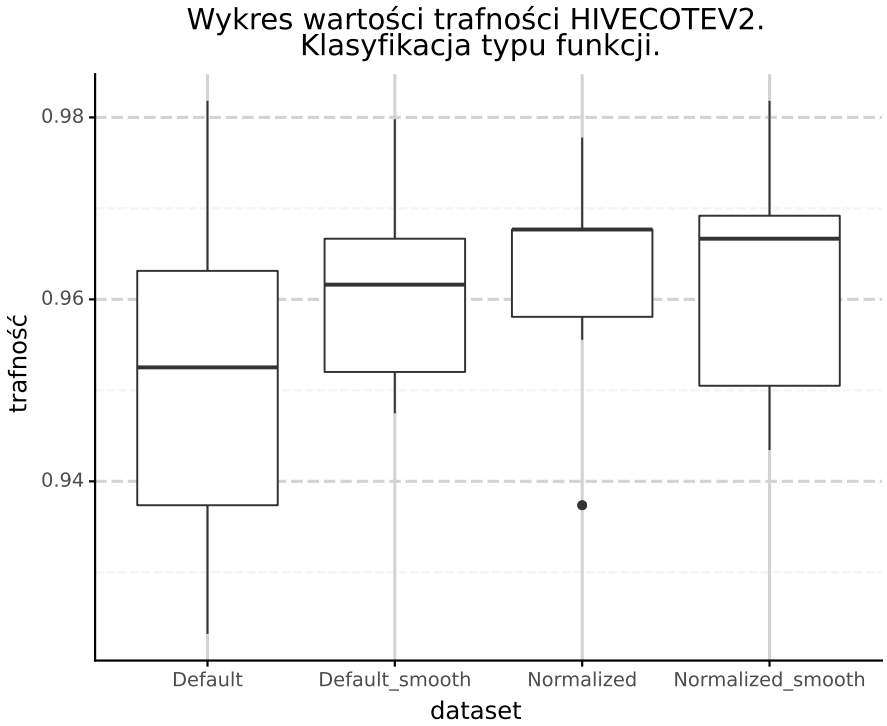
Wyniki metody HIVECOTEV2

HIVECOTEV2 to specjalistyczny algorytm służący do klasyfikacji szeregów czasowych. Nie uzyskał on tak wysokich wyników jak ROCKET, jednak wciąż są one imponujące. W jego przypadku najlepszy rezultat został uzyskany dla znormalizowanych zbiorów danych. *Normalized* jak i *Normalized_smooth* uzyskały średnią trafność i F-miarę równą 0.962. Klasyfikacja danych nieprzetworzonych również osiągnęła wysokie wartości dla trafności oraz F-miary. Tablica 5.2 przedstawia wyniki dla każdego zestawu danych.

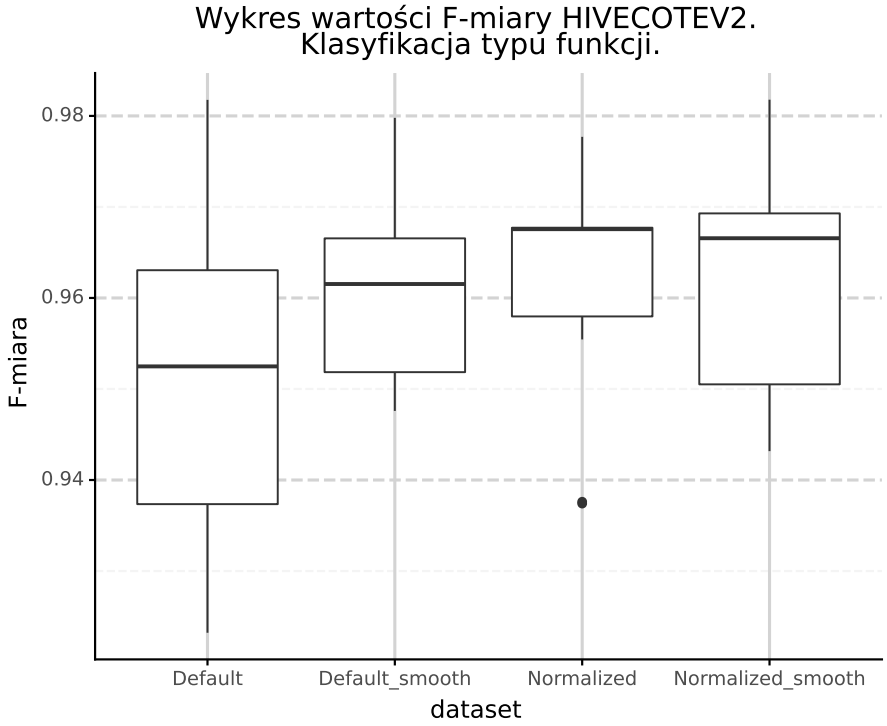
Tablica 5.2: Wyniki eksperymentów algorytmu HIVECOTEV2 dla klasyfikacji typów funkcji

zestaw danych (dataset)	trafność	f-miara	czas przetwarzania (s)
Default	0.951	0.951	938.365
Default_smooth	0.961	0.961	788.086
Normalized	0.962	0.962	848.660
Normalized_smooth	0.962	0.962	807.602

W przeciwieństwie do algorytmu ROCKET, normalizacja danych wpłynęła pozytywnie na klasyfikacje. Należy również wspomnieć, że algorytm HIVECOTEV2 podczas przeprowadzania eksperymentu został ograniczony czasowo przy pomocy jednego z parametrów. Ustawiony on został na pięć minut. Brak ograniczenia powodował zdecydowanie dłuższy czas wykonywania, który przekraczał kilka godzin bez zwrócenia wyników. Prawdopodobnie miało to wpływ na wyniki, jednak nawet przy limicie, te rezultaty są godne podziwu. Czas przetwarzania był ok. dziesięć razy dłuższy niż w przypadku algorytmu ROCKET. Ponownie losowość nie miała większego wpływu na wyniki, co można zauważyć na wykresach pudełkowych przedstawionych na rysunku 5.31.



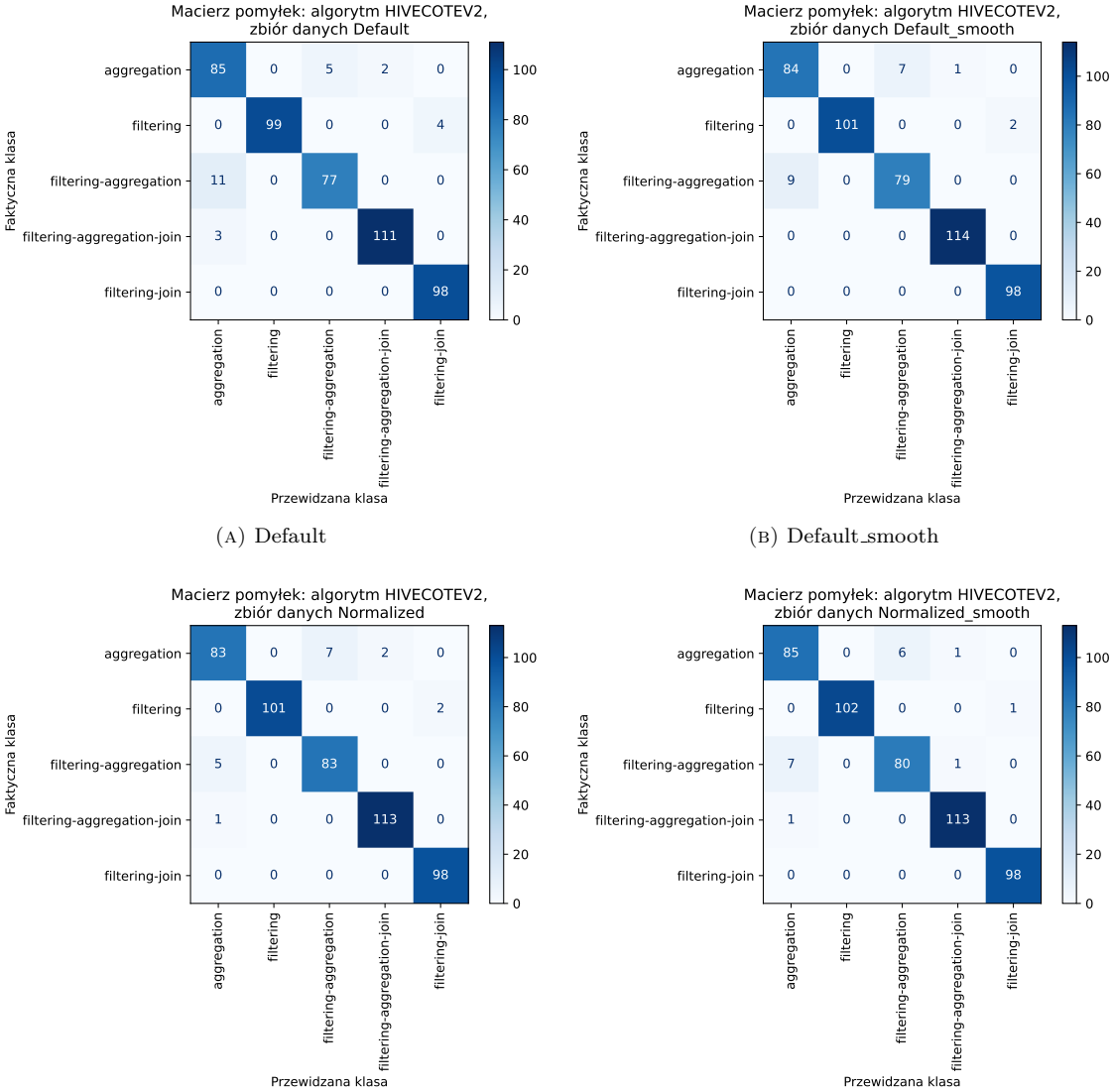
(A) Trafność



(B) F-miara

Rysunek 5.31: Wykresy pudełkowe trafności oraz F-miary dla algorytmu HIVECOTEV2.

Na rysunku 5.32, przedstawiającym macierze pomyłek, powtarza się trudność klasyfikacji klasy aggregation. Najczęściej algorytm myli ją z klasą filtering-aggregation. Poza tym pojawiły się pojedyncze pomyłki dla pozostałych funkcji. Potwierdza to tezę, że istnieje silne podobieństwo w ramach danych typów funkcji, umożliwiające tak trafne rezultaty badanych algorytmów.



(A) Default

(B) Default_smooth

(C) Normalized

(D) Normalized_smooth

Rysunek 5.32: Macierze pomyłek dla każdego zbioru danych algorytmu HIVECOTEV2

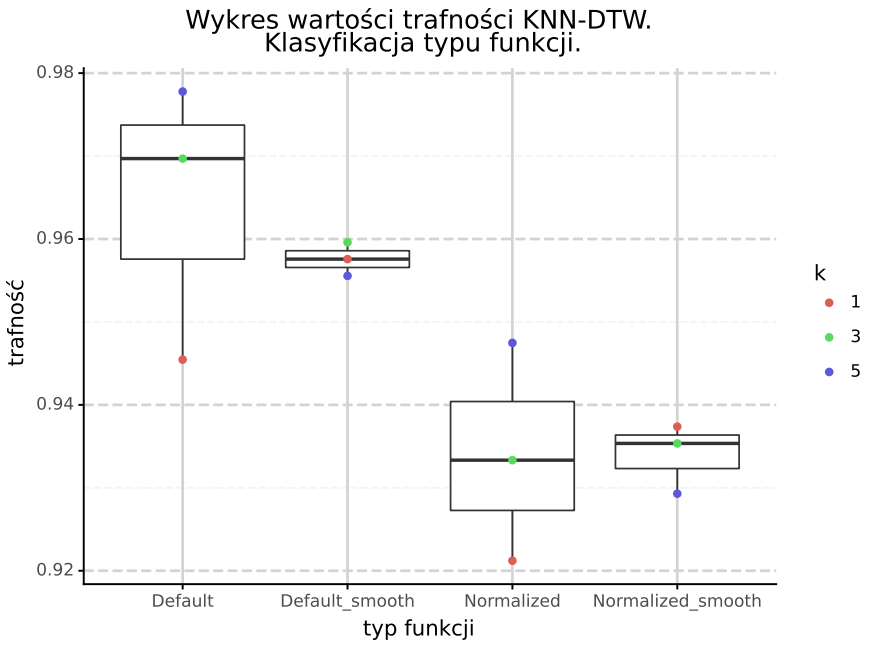
Wyniki metody KNN-DTW

KNN, czyli "K najbliższych sąsiadów", to klasyczny algorytm klasyfikujący, który został wybrany, ponieważ bada on bezpośrednio podobieństwo między poszczególnymi instancjami szeregów. W tym przypadku funkcje miary podobieństwa pełni algorytm DTW. Z względu na to, że nie ma możliwości ustawienia parametru sterującego losowością, powtórzenia przetwarzania zostały wykonane z różnymi wartościami parametru k. Wyznacza on liczbę sąsiadów braną pod uwagę podczas klasyfikacji. Wybrane wartości to 1, 3 i 5. Uruchomiony w ten sposób algorytm uzyskał trafność i F-miarę równą średnio 0.964 dla zestawu *Default*. Dla reszty zbiorów wyniki były równie wysokie. Tablica 5.3 pokazuje zbiorcze rezultaty dla badanego algorytmu.

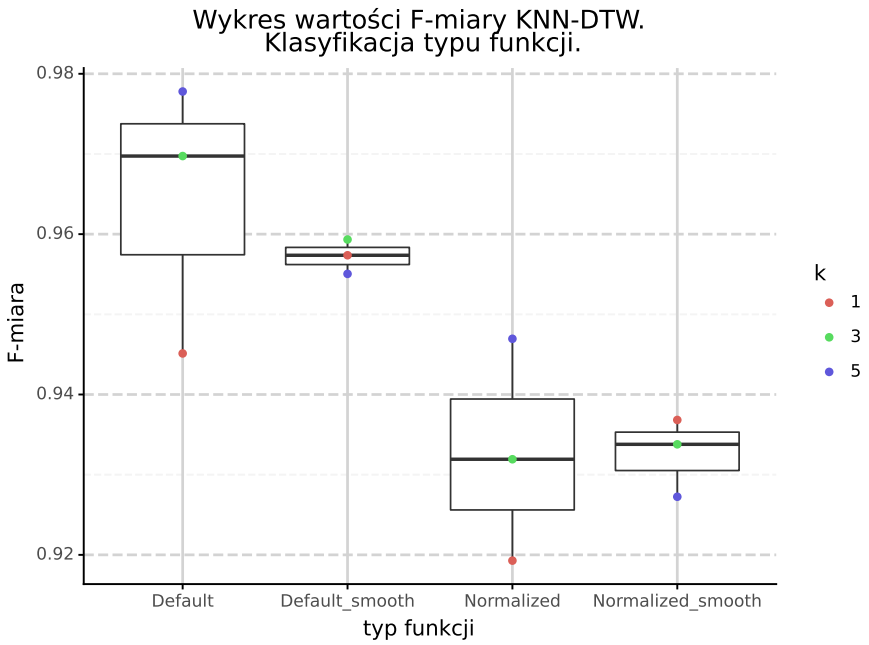
Tablica 5.3: Wyniki eksperymentów algorytmu KNN-DTW dla klasyfikacji typów funkcji

zestaw danych (dataset)	trafność	f-miara	czas przetwarzania (s)
Default	0.964	0.964	15843.151
Default_smooth	0.958	0.957	16314.820
Normalized	0.934	0.933	16356.924
Normalized_smooth	0.934	0.933	15196.359

Ponownie kolejny najlepsze wyniki algorytm uzyskał pracując na danych nieprzetworzonych (brak normalizacji oraz wygładzania). Normalizacja pogorszyła wynik o 0.03 w stosunku do najlepszego wyniku. To klasyczne podejście okazało się bardzo czasochłonne, gdyż średnio zajęło ok. pięciu godzin dla każdego uruchomienia. Jest to bardzo duża różnica, szczególne w porównaniu z najszybszym przetwarzaniem jakie osiągnęła metoda ROCKET (średnio ok. minute). Na rysunku 5.33 można zaobserwować, że liczba sąsiadów miała nieduży wpływ na wyniki. Różnica w rezultatach była bardziej widoczna między zbiorami danych.



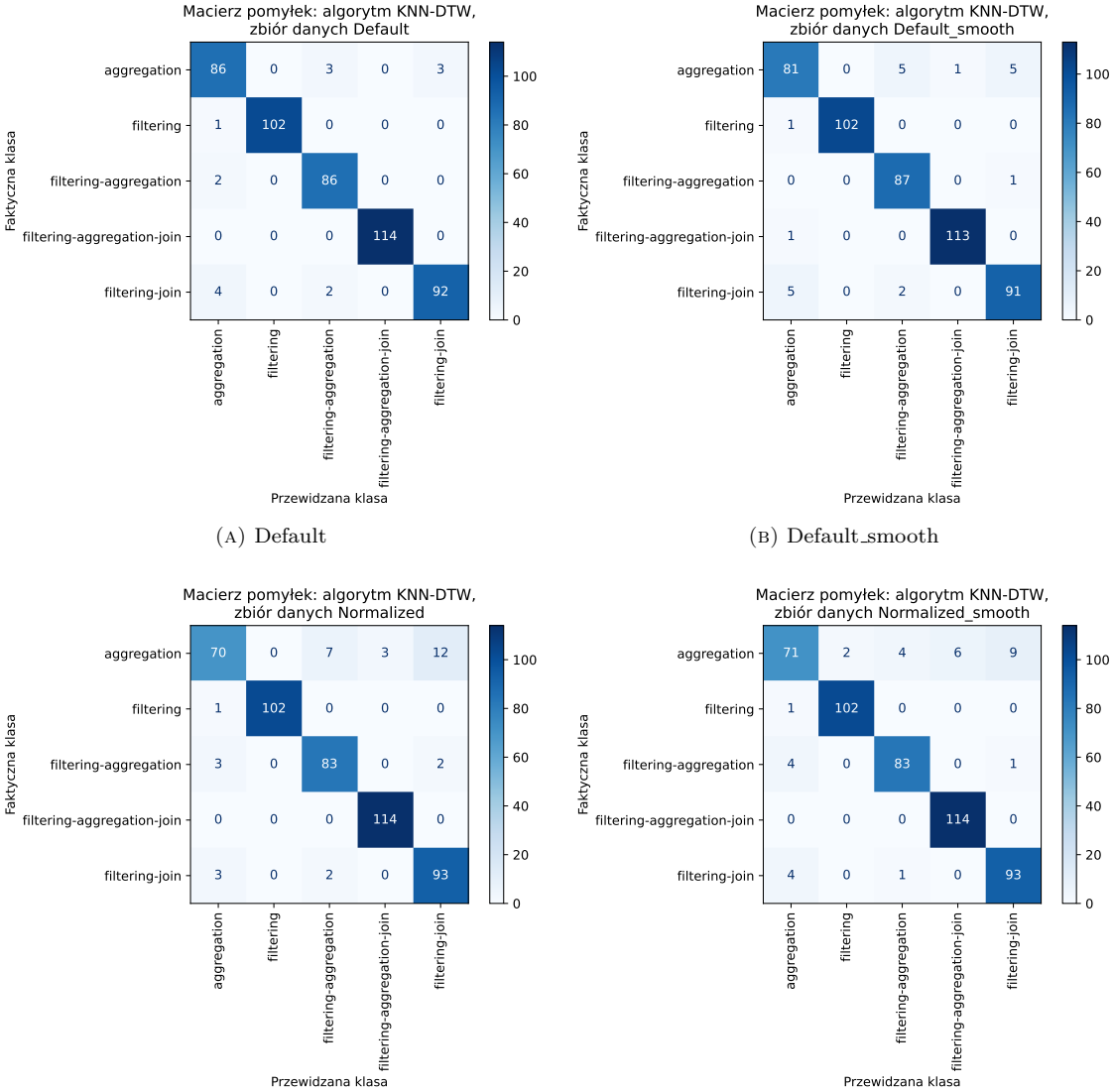
(A) Trafność



(B) F-miara

Rysunek 5.33: Wykresy pudełkowe trafności oraz F-miary dla algorytmu KNN-DTW.

W przypadku algorytmu KNN, wykorzystującego DTW jako miarę prawdopodobieństwa, najczęstsze błędy popełniał przy klasyfikacji typu aggregation. Na macierzach pomyłek widocznych na rysunku 5.34 można zauważyć, że algorytm najczęściej mylił agregacje z klasami filtering-aggregation (podobnie jak w przypadku poprzednich algorytmów). Filtering-join również był często mylony z klasą aggregation, co może sugerować pewne podobieństwo w szeregach czasowych reprezentującymi te dwa typy przetwarzania.



(A) Default

(B) Default_smooth

(C) Normalized

(D) Normalized_smooth

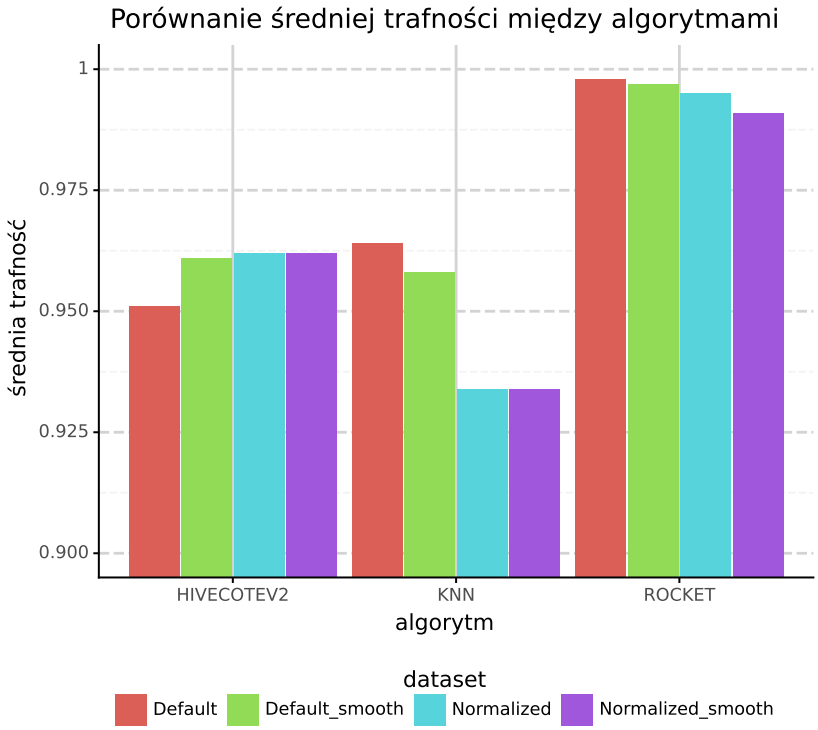
Rysunek 5.34: Macierze pomyłek dla każdego zbioru danych algorytmu KNN-DTW

Porównanie i podsumowanie

Klasyfikacja miała na celu sprawdzenie, czy jest możliwe rozróżnienie typów funkcji oraz czy występuje podobieństwo między przebiegami zużycia zasobów na maszynie podczas przetwarzania tych samych operacji. Wszystkie trzy badane metody wykazały, że identyfikacja konkretnego typu nie jest trudna. Szczególnie ciekawe są rezultaty algorytmu KNN-DTW, ponieważ jako jedyny korzysta on z miary podobieństwa. Jego najlepsze średnie wyniki porównywalne są z resztą algorytmów, które specjalizują się w klasyfikacji szeregów czasowych. Można przypuszczać, że szeregi reprezentujące zużycie zasobów komputera podczas przetwarzania danych są podobne, jeżeli do przetwarzania wykorzystywane są te same operacje.

Algorytmem, który uzyskał najlepsze wyniki, ostatecznie okazał się ROCKET. Dla tego podejścia wartości trafności i F-miary osiągały średnio 0.995. Przetwarzanie również było najkrótsze. Pojedyncze eksperymenty ROCKET wykonywały się średnio ok. 64 sekundy. Warto zauważyć, że wyniki dla zbioru danych *Default* były średnio najlepsze (z wyjątkiem HIVECOTEV2, gdzie najkorzystniejszy okazał się zestaw *Normalized*). Wygładzanie danych nie miało większego w wpływu na

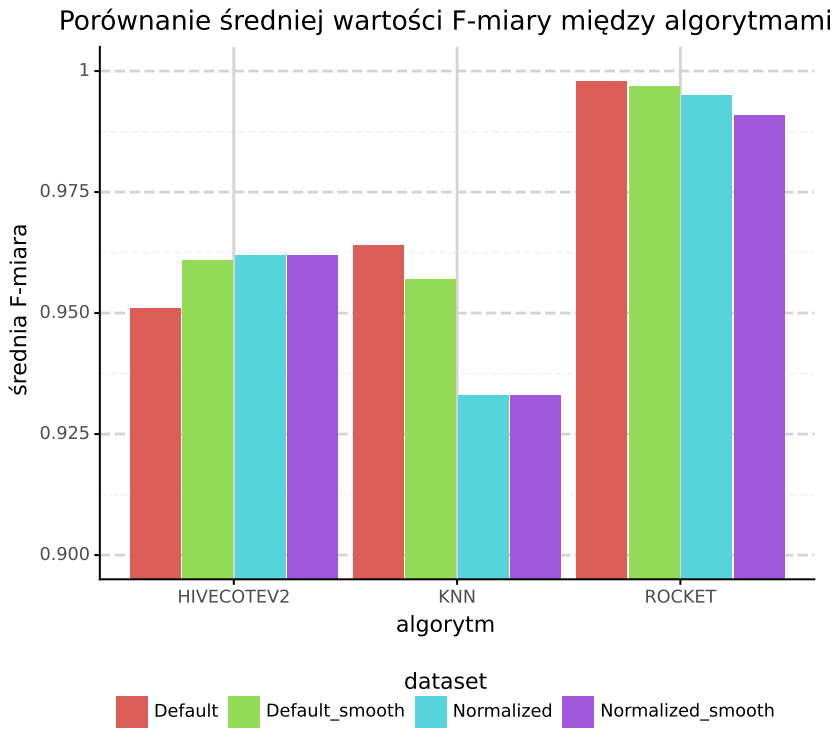
wyniki, a czasem nawet je pogarszało. Na wykresach od 5.35 do 5.37 pokazano zbiorcze rezultaty, z podziałem na zbiory danych i algorytmy.



Rysunek 5.35: Porównanie średniej trafności z dziesięciu eksperymentów ROCKET oraz HIVECOTEV2 i trzech eksperymentów KNN. Badane zestawy danych zawierały szeregi RAM i CPU, przedstawiające przetwarzanie danych o rozmiarze 1GB oraz 2GB.

Tablica 5.4: Zestawienie średniej trafności widocznej na wykresie 5.35

zestaw danych (dataset)	HIVECOTEV2	KNN	ROCKET
Default	0.951	0.964	0.998
Default_smooth	0.961	0.958	0.997
Normalized	0.962	0.934	0.995
Normalized_smooth	0.962	0.934	0.991



Rysunek 5.36: Porównanie średniej F-Miary z dziesięciu eksperymentów ROCKET oraz HIVECOTEV2 i trzech eksperymentów KNN. Badane zestawy danych zawierały szeregi RAM i CPU, przedstawiające przetwarzanie danych o rozmiarze 1GB oraz 2GB.

Tablica 5.5: Zestawienie średniej F-miary widocznej na wykresie 5.36

zestaw danych (dataset)	HIVECOTEV2	KNN	ROCKET
Default	0.951	0.964	0.998
Default_smooth	0.961	0.957	0.997
Normalized	0.962	0.933	0.995
Normalized_smooth	0.962	0.933	0.991

Rysunek 5.37: Porównanie średniego czasu przetwarzania z wszystkich eksperymentów ROCKET, HIVECOTEV2 i KNN niezależnie od zbioru danych. Badane zestawy danych zawierały szeregi RAM i CPU, przedstawiające przetwarzanie danych o rozmiarze 1GB oraz 2GB. Parametry komputera wykonującego pomiary: CPU-AMD Ryzen 5 2600x (3.6 GHz, 6 rdzeni, 12 wątków), RAM-2x8GB (3200MHz)

Tablica 5.6: Zestawienie średnich czasów przetwarzania widocznych na wykresie 5.37

algorytm	średni czas przetwarzania (minuty)	odchylenie standardowe (minuty)
HIVECOTEV2	14.09	1.11
KNN	265.46	9.01
ROCKET	1.07	0.04

5.2 Podobieństwo wolumenów danych

5.2.1 Algorytmy badające podobieństwo

Aby sprawdzić wpływ rozmiaru danych na podobieństwo szeregów czasowych, obliczono procentową różnicę między wynikami algorytmów podobieństwa przebiegów zużycia zasobów przy przetwarzaniu danych o rozmiarze 1GB, a 2GB. Porównanie zostało wykonane dla każdego typu funkcji z każdym. Jeśli przyjąć avg1GB oraz avg2GB jako średnią odległość między danymi typami funkcji, na danych o rozmiarze odpowiednio 1GB i 2GB, obliczenie tej różnicy można opisać wzorem 5.1

$$\frac{avg1GB - avg2GB}{avg1GB} * 100\% \tag{5.1}$$

Do oceny wpływu rozmiaru danych na możliwość znalezienia jaki typ operacji został wykorzystany przez funkcje przedstawioną jako szereg czasowy, najbardziej znaczące są różnice średniej odległości dla tych samych typów UDF (np. średniej odległości agregacji od innej agregacji).

Przykładowo, wyliczono różnicę procentową dla średniej odległości euklidesowej pomiędzy szeregami przedstawiającymi agregację na danych zużycia CPU o rozmiarze 1GB i średniej na danych zużycia CPU o rozmiarze 2GB. Aby uzyskać różnicę obliczono średnią odległość, osobno dla danych o rozmiarze 1GB i osobno dla 2GB. Wyniosła ona odpowiednio 1119.27 dla 1GB i 1148.18 dla 2GB. Po podstawieniu do wzoru różnica procentowa to 2.58%. Po obliczeniu różnic dla pozostałych

typów uzyskano tablicę 5.7.

$$\frac{1119.27 - 1148.18}{1119.27} * 100\% = 2.58\%$$

Dodatkowo pokazano również procentową różnicę między wynikami dla danych o różnych wielkościach, a wynikami dla danych o tych samym rozmiarze. Jeśli przyjąć avgDifferent jako średnią odległość między operacjami na danych o różnych rozmiarach i avgSame jako średnią odległość między operacjami na danych o tych samych rozmiarach, obliczenie tej różnicy można opisać wzorem 5.2

$$\frac{avgDifferent - avgSame}{avgDifferent} * 100\% \tag{5.2}$$

Dla przykładu obliczono różnicę procentową dla odległości pomiędzy agregacjami, które zostały wykonane na danych CPU o tym samym rozmiarze i dla odległości pomiędzy agregacjami, które uzyskano porównując szeregi dla danych CPU o rozmiarze 1GB i dla danych o wielkości 2GB. na danych CPU. W tym celu obliczono średnią odległość osobno dla porównań szeregów, przy których jeden z nich powstał przy użyciu danych CPU o wielkości 1GB, a drugi przy użyciu danych o rozmiarze 2GB i osobno dla porównań agregacji na danych o tym samym rozmiarze. Wyniosła ona odpowiednio 1170.02 dla różniących się rozmiarów i 1137.47 dla tych samych. Podstawienie do wzoru dało 2.78%. Po obliczeniu różnic dla pozostałych typów uzyskano tablicę 5.8.

$$\frac{1170.02 - 1137.47}{1170.02} * 100\% = 2.78\%$$

Odległość Euklidesowa

W tablicy 5.7 przedstawiono procentową różnicę między odległościami euklidesowymi dla danych przedstawiających zużycie CPU o wielkości 1GB, a odległościami dla danych przedstawiających zużycie CPU o wielkości 2GB. W tablicy 5.8 można zobaczyć różnicę między odległościami euklidesowymi dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - również dla zużycia CPU. Można zauważyć, że różnice nie są znaczące - w pierwszym przypadku nie przekraczają 9%, a w drugim 4%. Dla większości porównań są to liczby dużo mniejsze. Przy porównywaniu 1GB z 2GB nie ma stałej zależności co do tego, który z rozmiarów zwraca większą odległość. W tablicy 5.8 można zauważyć jednak, że w 80% porównania danych o różnych wielkościach zwrócą większą odległość, niż porównania danych o tym samym rozmiarze, co jest zgodne z intuicją i wiedzą na temat działania odległości euklidesowej.

Tablica 5.7: Procentowa różnica między odległościami euklidesowymi dla danych o wielkości 1GB, a odległościami dla danych o wielkości 2GB - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-2.58%	0.97%	3.5%	2.43%	-1.52%
filtering	0.97%	2.11%	5.72%	4.95%	-0.74%
filtering-aggregation	3.5%	5.72%	5.59%	3.73%	3.64%
filtering-aggregation-join	2.43%	4.95%	3.73%	3.81%	4.32%
filtering-join	-1.52%	-0.74%	3.64%	4.32%	-8.61%

Tablica 5.8: Procentowa różnica między odległościami euklidesowymi dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	2.78%	0.53%	1.49%	1.36%	0.35%
filtering	0.53%	0.54%	-0.24%	0.27%	0.23%
filtering-aggregation	1.49%	-0.24%	3.7%	2.4%	-0.24%
filtering-aggregation-join	1.36%	0.27%	2.4%	3.93%	0.11%
filtering-join	0.35%	0.23%	-0.24%	0.11%	2.54%

Tablice 5.9 i 5.10 przedstawiają te same porównania tylko dla danych przedstawiających zużycie pamięci RAM. Liczby są tu znacznie większe i sugerują duże różnice między rozmiarami danych. Jednak patrząc na rysunek 5.7 i pamiętając, że zużycie pamięci RAM może służyć jedynie do rozróżnienia filtracji oraz filtracji-join od pozostałych funkcji, których odległości między sobą są kilkukrotnie mniejsze od pozostałych można wywnioskować, że na odróżnienie filtracji i filtracji-join od innych typów funkcji, rozmiar danych nie powinien znacząco wpływać.

Tablica 5.9: Procentowa różnica między odległościami euklidesowymi dla danych o wielkości 1GB, a odległościami dla danych o wielkości 2GB - zużycie RAM

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-70.47%	-52.47%	-76.34%	-91.12%	-57.04%
filtering	-52.47%	-60.01%	-36.93%	-30.23%	-30.61%
filtering-aggregation	-76.34%	-36.93%	-50.46%	-27.45%	-41.36%
filtering-aggregation-join	-91.12%	-30.23%	-27.45%	-14.6%	-33.79%
filtering-join	-57.04%	-30.61%	-41.36%	-33.79%	-68.31%

Tablica 5.10: Procentowa różnica między odległościami euklidesowymi dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - zużycie RAM

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	51.6%	-1.23%	47.03%	40.14%	1.29%
filtering	-1.23%	29.06%	-2.28%	-0.99%	20.52%
filtering-aggregation	47.03%	-2.28%	54.4%	49.5%	0.23%
filtering-aggregation-join	40.14%	-0.99%	49.5%	53.32%	0.83%
filtering-join	1.29%	20.52%	0.23%	0.83%	23.86%

Dynamic Time Warping

W tablicy 5.11 przedstawiono procentową różnicę między odległościami DTW dla danych przedstawiających zużycie CPU o wielkości 1GB, a odległościami dla danych przedstawiających zużycie CPU o wielkości 2GB. W tablicy 5.12 można zobaczyć różnicę między odległościami DTW dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - również dla zużycia CPU. Różnice w pierwszym przypadku nie przekraczają 16%, a w drugim 6%. Są to różnice większe niż przy odległości euklidesowej, jednak również różnice między typami (rysunek 5.11) były

bardziej znaczące. Można wywnioskować, że Dynamic Time Warping radzi sobie ze skalowaniem czasu. Różnice między wynikami dla poszczególnych rozmiarów są jednak dość wyraźne.

Tablica 5.11: Procentowa różnica między odległościami DTW dla danych o wielkości 1GB, a odległościami dla danych o wielkości 2GB - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-11.56%	-9.46%	-10.15%	1.32%	-15.51%
filtering	-9.46%	3.86%	-4.93%	1.12%	2.2%
filtering-aggregation	-10.15%	-4.93%	-4.18%	1.5%	-7.04%
filtering-aggregation-join	1.32%	1.12%	1.5%	0.38%	-3.76%
filtering-join	-15.51%	2.2%	-7.04%	-3.76%	-2.37%

Tablica 5.12: Procentowa różnica między odległościami DTW dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	6.1%	0.58%	5.36%	2.48%	0.94%
filtering	0.58%	1.35%	-0.22%	0.26%	0.08%
filtering-aggregation	5.36%	-0.22%	4.77%	2.07%	0.4%
filtering-aggregation-join	2.48%	0.26%	2.07%	2.42%	0.2%
filtering-join	0.94%	0.08%	0.4%	0.2%	0.62%

Tablice 5.13 i 5.14 zawierają te same porównania co dwie poprzednie, tylko dla danych przedstawiających zużycie pamięci RAM. W tym wypadku różnice między porównywaniem 1GB, a 2GB są ogromne.

Tablica 5.13: Procentowa różnica między odległościami DTW dla danych o wielkości 1GB, a odległościami dla danych o wielkości 2GB - zużycie RAM

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-114.93%	-115.34%	-151.59%	-91.13%	-77.84%
filtering	-115.34%	-67.11%	-119.05%	-104.78%	-61.54%
filtering-aggregation	-151.59%	-119.05%	-603.72%	-440.61%	-88.25%
filtering-aggregation-join	-91.13%	-104.78%	-440.61%	-92.7%	-66.09%
filtering-join	-77.84%	-61.54%	-88.25%	-66.09%	-55.4%

Tablica 5.14: Procentowa różnica między odległościami DTW dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - zużycie RAM

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-18.85%	-12.47%	-18.85%	-11.44%	-4.92%
filtering	-12.47%	-2.39%	-1.04%	-12.17%	-1.57%
filtering-aggregation	-18.85%	-1.04%	-4.63%	-26.17%	1.82%
filtering-aggregation-join	-11.44%	-12.17%	-26.17%	8.03%	-7.33%
filtering-join	-4.92%	-1.57%	1.82%	-7.33%	-0.16%

Longest Common Subsequence

W tablicy 5.15 przedstawiono procentową różnicę między odległością Longest Common Subsequence dla danych przedstawiających zużycie CPU o wielkości 1GB, a odległością dla danych przedstawiających zużycie CPU o wielkości 2GB. W tablicy 5.16 można zobaczyć różnicę między odległością Longest Common Subseqence dla danych o różnych wielkościach, a odległością dla danych o tej samej wielkości - również dla zużycia CPU. Dla tablicy 5.15 różnice są wyraźne, jednak nie mają wpływu na odróżnienie filtracji i filtracji-join od reszty. Różnice w tablicy 5.16 są mniej znaczące.

Tablica 5.15: Procentowa różnica między odległością LCSS dla danych o wielkości 1GB, a odległościami dla danych o wielkości 2GB - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-54.0%	-55.22%	-46.16%	-22.36%	-57.27%
filtering	-55.22%	-17.82%	-31.26%	-18.95%	-7.67%
filtering-aggregation	-46.16%	-31.26%	-30.85%	-17.96%	-33.18%
filtering-aggregation-join	-22.36%	-18.95%	-17.96%	-15.47%	-20.08%
filtering-join	-57.27%	-7.67%	-33.18%	-20.08%	-12.0%

Tablica 5.16: Procentowa różnica między odległością LCSS dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	9.18%	-0.08%	10.19%	3.71%	2.03%
filtering	-0.08%	4.11%	-0.43%	-0.05%	0.68%
filtering-aggregation	10.19%	-0.43%	9.83%	4.82%	1.2%
filtering-aggregation-join	3.71%	-0.05%	4.82%	10.9%	0.85%
filtering-join	2.03%	0.68%	1.2%	0.85%	2.39%

Tablice 5.17 i 5.18 zawierają te same porównania co dwie poprzednie, jednakże tym razem z użyciem pamięci RAM. Wartości są tu duże i mocno różniące się od siebie, jednak nadal umożliwiające odróżnienie filtracji i filtracji-join od reszty. Średnia odległość jest zawsze mniejsza dla danych o rozmiarze 1GB, niż dla danych o rozmiarze 2GB.

Tablica 5.17: Procentowa różnica między odległością LCSS dla danych o wielkości 1GB, a odległościami dla danych o wielkości 2GB - zużycie RAM

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	-137.51%	-148.07%	-163.66%	-154.4%	-159.6%
filtering	-148.07%	-100.85%	-119.41%	-78.2%	-67.38%
filtering-aggregation	-163.66%	-119.41%	-345.69%	-230.37%	-131.18%
filtering-aggregation-join	-154.4%	-78.2%	-230.37%	-68.3%	-85.28%
filtering-join	-159.6%	-67.38%	-131.18%	-85.28%	-90.63%

Tablica 5.18: Procentowa różnica między odległością LCSS dla danych o różnych wielkościach, a odległościami dla danych o tej samej wielkości - zużycie CPU

typ funcji typ funcji	aggregation	filtering	filtracja-agregacja	filtracja-agregacja -join	filtering-join
aggregation	43.33%	0.33%	44.09%	36.16%	4.25%
filtering	0.33%	29.52%	-0.73%	0.15%	22.7%
filtering-aggregation	44.09%	-0.73%	53.89%	47.7%	3.75%
filtering-aggregation-join	36.16%	0.15%	47.7%	60.29%	3.72%
filtering-join	4.25%	22.7%	3.75%	3.72%	22.13%

Podsumowanie

Różnice między odległościami były znacznie większe, gdy porównywano odległości dla danych o rozmiarze 1GB i dla danych o rozmiarze 2GB. Z tego wynika, że w miarę możliwości wskazane byłoby przechowywanie i porównywanie szeregów czasowych powstałych z przetwarzania danych o różnorodnej wielkości. Warto też zauważyć, że eksperymenty zostały wykonane dla jedynie dwóch rozmiarów danych.

5.2.2 Algorytmy uczenia maszynowego

Wszystkie eksperymenty w niniejszej sekcji zostały wykonane przy pomocy notatnika *9.2_run_machine_learning_size_experiments.ipynb*, który tworzy pliki z rezultatami klasyfikacji. Aby zachować spójność z poprzednimi eksperymentami wykorzystującymi uczenie maszynowe, poniższe eksperymenty również były powtórzone dziesięć razy w przypadku algorytmów ROCKET oraz HIVECOTEV2, a dla KNN przetwarzanie powtórzono dla trzech różnych wartości parametru k (ponownie wykorzystując wartości 1, 3 oraz 5). Na podstawie wyników z sekcji 5.1.2 algorytm został uruchomiony używając danych ze zbioru *"Default"* (dane nieprzetworzone). Ten zestaw danych został wybrany, ponieważ uczenie maszynowe działające na tym zbiorze danych osiągało średnio najlepsze rezultaty. Każdy z dostępnych typów operacji zbadano osobno, poprzez klasyfikacje szeregów reprezentujących przetwarzanie dwóch dostępnych rozmiarów danych; 1GB oraz 2 GB. Każdy z badanych algorytmów miał możliwość wykorzystania wielowymiarowych szeregów czasowych. Z tego powodu **do klasyfikacji wykorzystane są przebiegi zużycia zarówno CPU jak i RAM.** Wizualizacja niniejszych eksperymentów została wykonana przy pomocy notatnika *9.4_results_for_size_machine_learning_experiments.ipynb*.

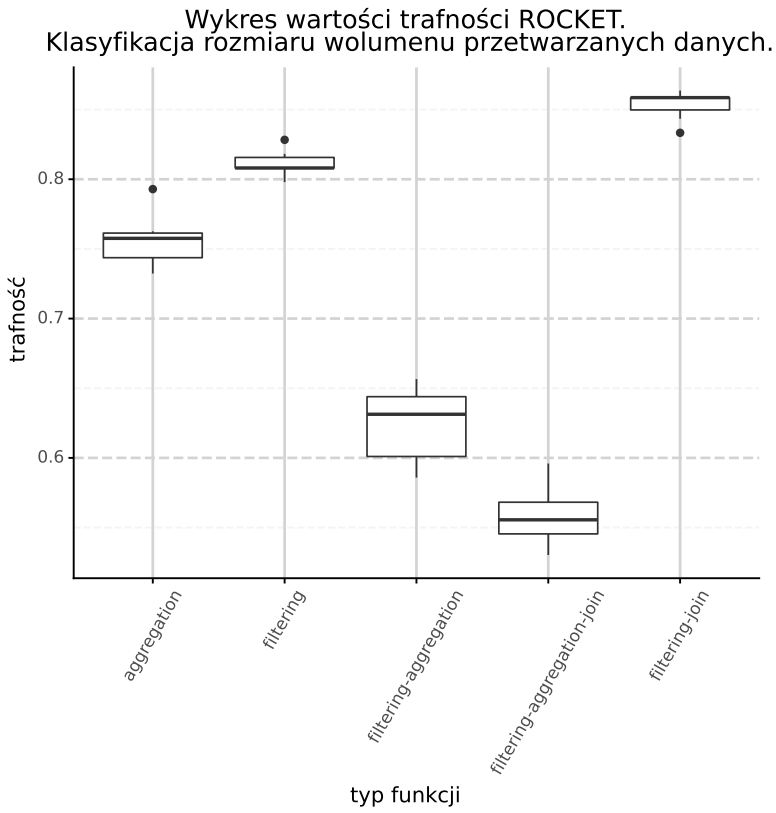
Wyniki metody ROCKET

Algorytm ROCKET klasyfikując wolumeny danych uzyskał różne wyniki zależnie od typu przetwarzania. Najlepiej rozróżniał wolumeny w przypadku funkcji typu filtering-join osiągając trafność i F-miarę równą średnio 0.854. Najgorzej z kolei poradził sobie z typem filtering-aggregation-join z wynikami 0.560 dla trafności i 0.558 dla F-miary. W tablicy 5.19 zebrane są rezultaty dla wszystkich typów funkcji.

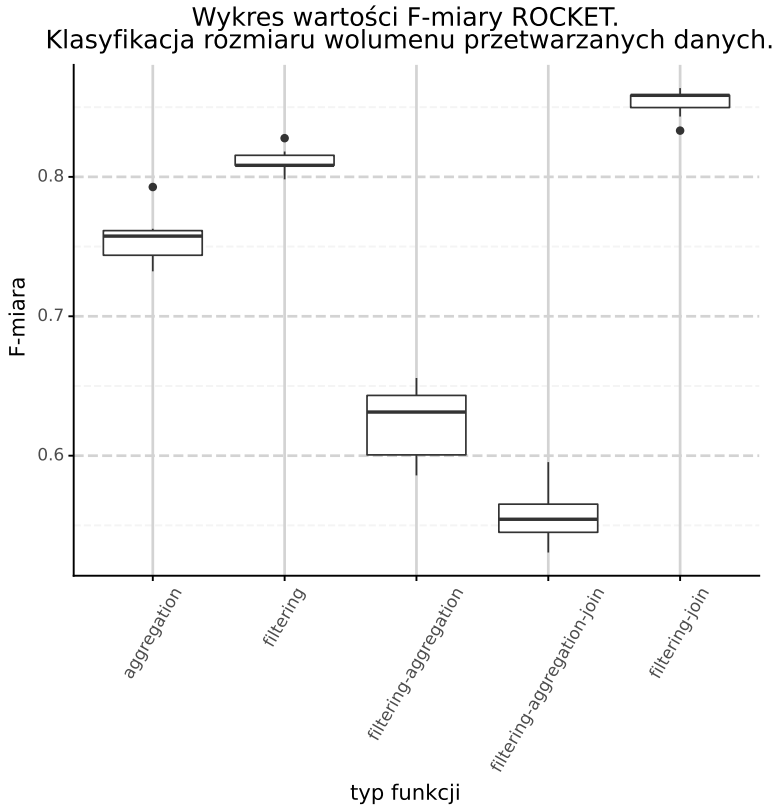
Tablica 5.19: Wyniki eksperymentów algorytmu ROCKET dla klasyfikacji wolumenów danych w ramach funkcji

typ funkcji	trafność	f-miara	czas przetwarzania (s)
aggregation	0.756	0.756	27.760
filtering	0.810	0.810	5.779
filtering-aggregation	0.624	0.624	23.967
filtering-aggregation-join	0.560	0.558	23.324
filtering-join	0.854	0.854	9.874

Wszystkie wyniki osiągają trafność powyżej 50%. Pokazuję to, że algorytm ROCKET radzi sobie z rozróżnianiem szeregów czasowych reprezentujących przetwarzanie 1GB oraz 2GB. Mogłoby to oznaczać, że w ramach tego samego typu funkcji (np. w ramach agregacji), szeregi, które reprezentują przetwarzanie różnych wolumenów danych są od siebie różne. Ponownie algorytm ten wykazuje krótki czas przetwarzania, średnio jest to ok. 18 sekund. Zaobserwowane, że dla krótszych przebiegów (funkcje typu filtering miały najkrótsze przebiegi) czas przetwarzania znacząco maleje. Wykresy pudełkowe 5.38 wskazują na to, że wyniki te były nieczułe na czynnik losowy.



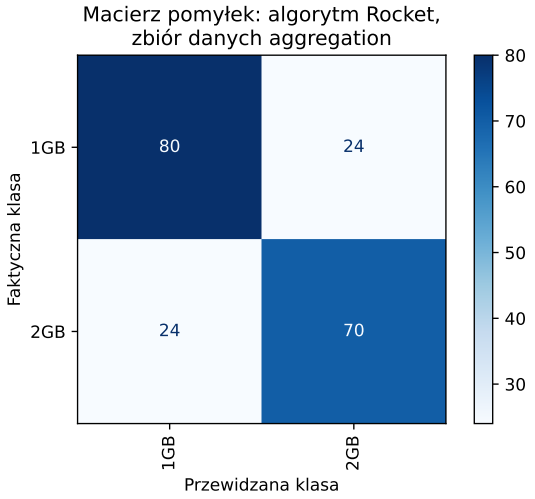
(A) Trafność



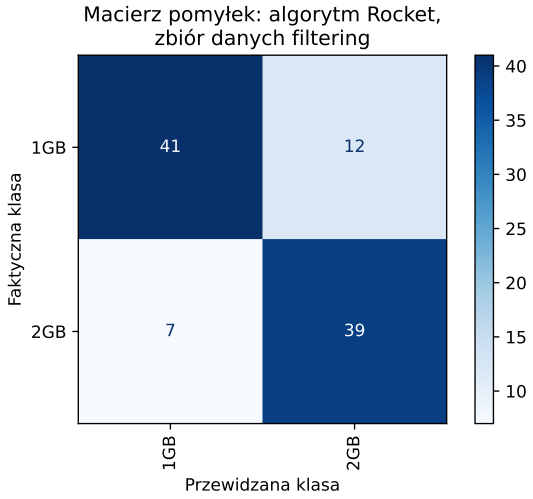
(B) F-miara

Rysunek 5.38: Wykresy pudełkowe trafności oraz F-miary dla algorytmu ROCKET.

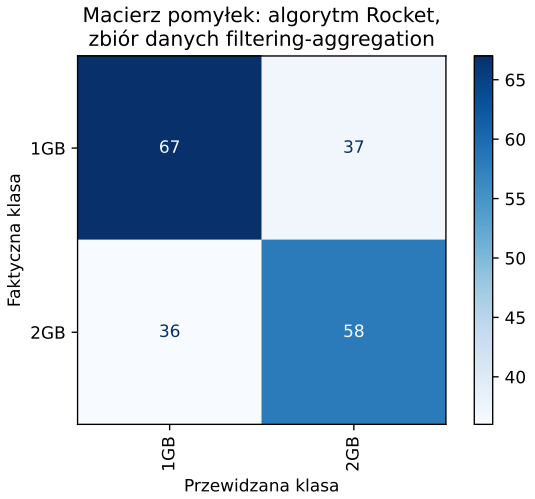
Rysunek 5.39 zawiera wszystkie macierze pomyłek badanych problemów. Warto zwrócić uwagę na, że pomyłki były albo równej liczności (przykładowo dla *aggregation* i *filtering-aggregation*) lub częściej popełniany był błąd klasyfikacji, gdzie niepoprawnie przypisana została klasa 2GB, kiedy faktycznie był przetwarzany 1GB.



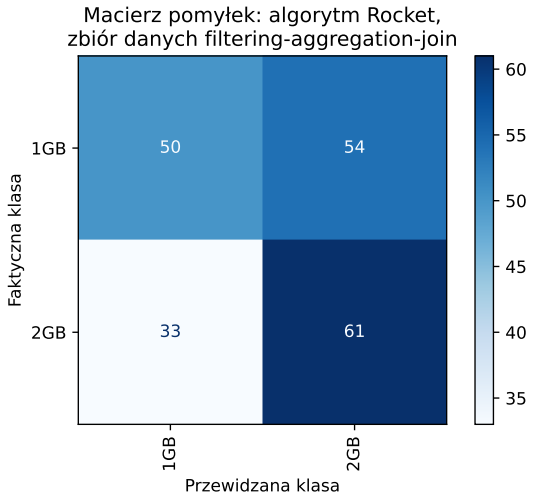
(A) Aggregation



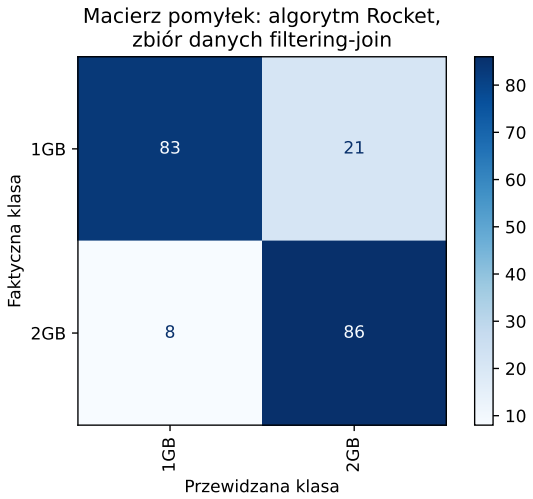
(B) Filtering



(C) Filtering-aggregation



(D) Filtering-aggregation-join



(E) Filtering-join

Rysunek 5.39: Macierze pomyłek dla każdego typu funkcji algorytmu ROCKET.

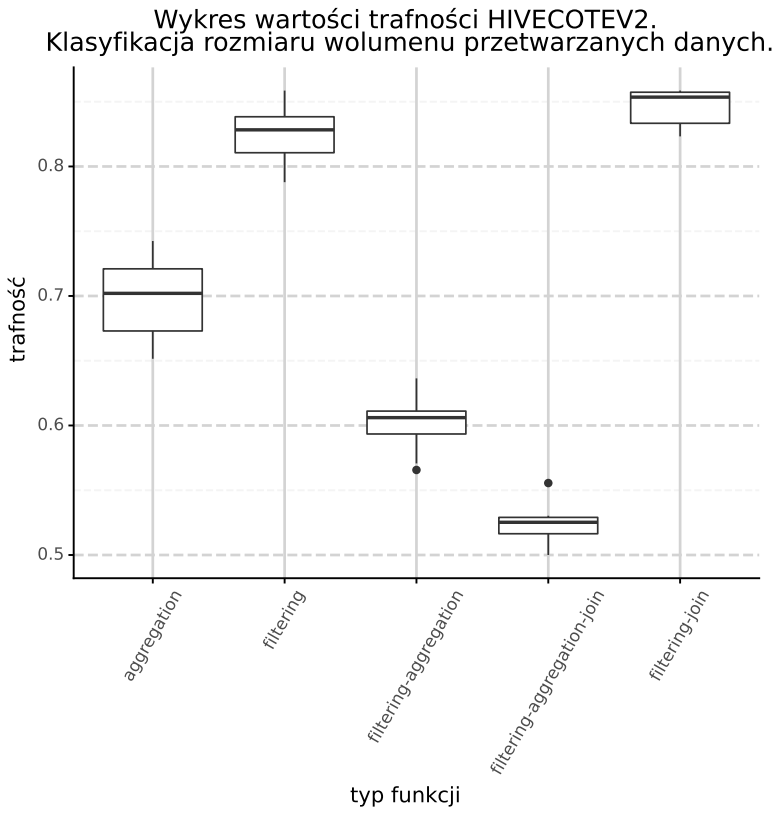
Wyniki metody HIVECOTEV2

HIVECOTEV2 uzyskał wyniki bardzo podobne do algorytmu ROCKET. Najwyższą wartość trafności oraz F-miary otrzymał dla typu filtering-join równą 0.845. Najniższe natomiast dla operacji filtering-aggregation-join w wysokości 0.523 dla trafności i 0.523 do F-miary. Wyniki klasyfikacji dla reszty typów funkcji są wypisane w tablicy 5.20.

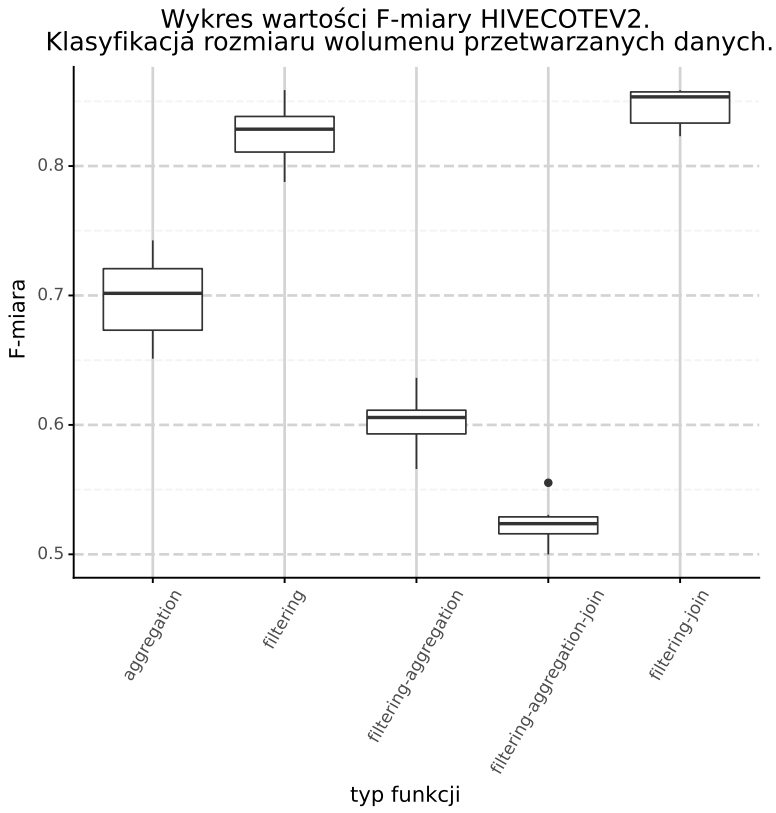
Tablica 5.20: Wyniki eksperymentów algorytmu HIVECOTEV2 dla klasyfikacji wolumenów danych w ramach funkcji

typ funkcji	trafność	f-miara	czas przetwarzania (s)
aggregation	0.699	0.699	670.338
filtering	0.826	0.826	293.617
filtering-aggregation	0.602	0.601	575.628
filtering-aggregation-join	0.523	0.523	612.062
filtering-join	0.845	0.845	468.517

W celu zachowania spójności między eksperymentami, HIVECOTEV2, tak jak podczas eksperymentów klasyfikacji typów (sekcja 5.1.2), został ograniczony czasowo. Mogło to mieć negatywny wpływ na jakość wyników. Po raz kolejny zauważono, że trudność klasyfikacji zależy od typu funkcji. Może to oznaczać, że część operacji jest bardziej czuła na rozmiar danych. Wszystkie wartości trafności są powyżej 50%, jest więc to kolejna metoda radząca sobie z klasyfikacją rozmiaru danych w ramach typów funkcji. Czas wykonywania się algorytmu HIVECOTEV2 również był istotnie zależny od długości. Krótsze szeregi przetwarzały się znacznie szybciej. Wykres pudełkowy 5.40 pokazuje niewielki wpływ losowości na wyniki. Oznacza to że algorytm HIVECOTEV2 działa stabilnie przy użyciu zebranych danych.



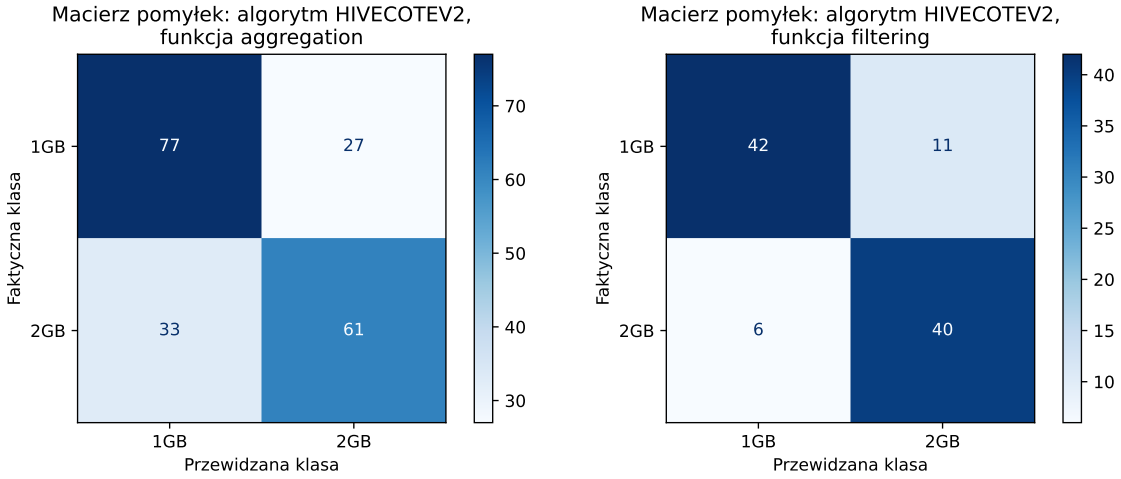
(A) Trafność



(B) F-miara

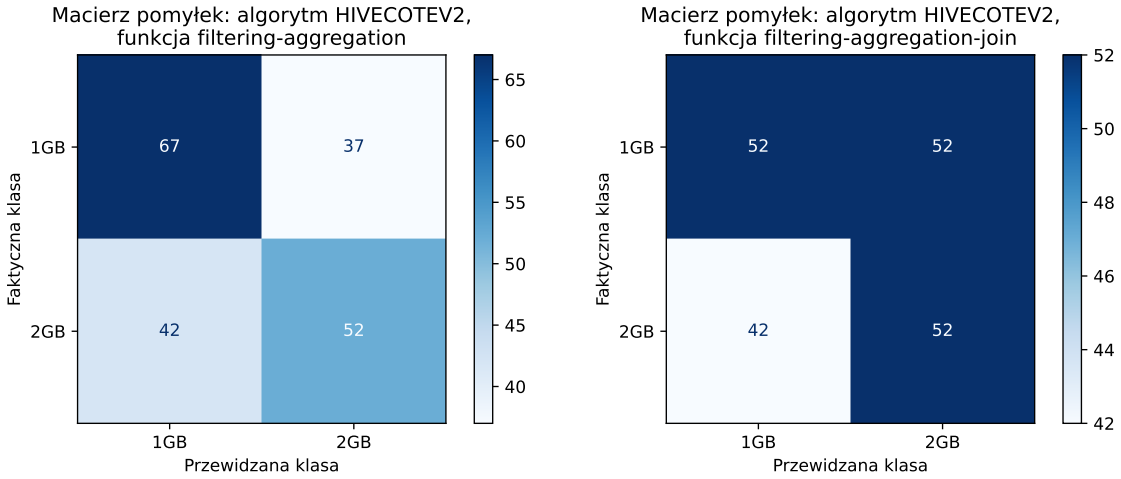
Rysunek 5.40: Wykresy pudełkowe trafności oraz F-miary dla algorytmu HIVECOTEV2.

Na rysunku 5.43 pokazane są macierze pomyłek po klasyfikacji. Z wyjątkiem funkcji typu filtering-aggregation, przeważają pomyłki, gdzie algorytm przypisał klasę 2GB do szeregu przetwarzającego 1GB.



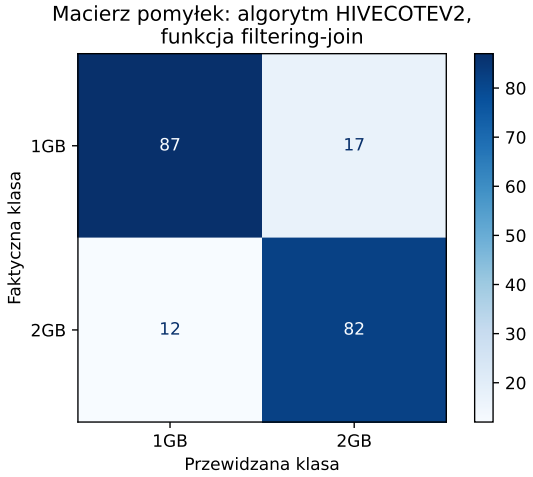
(A) Aggregation

(B) Filtering



(C) Filtering-aggregation

(D) Filtering-aggregation-join



(E) Filtering-join

Rysunek 5.41: Macierze pomyłek dla każdego typu funkcji algorytmu HIVECOTEV2.

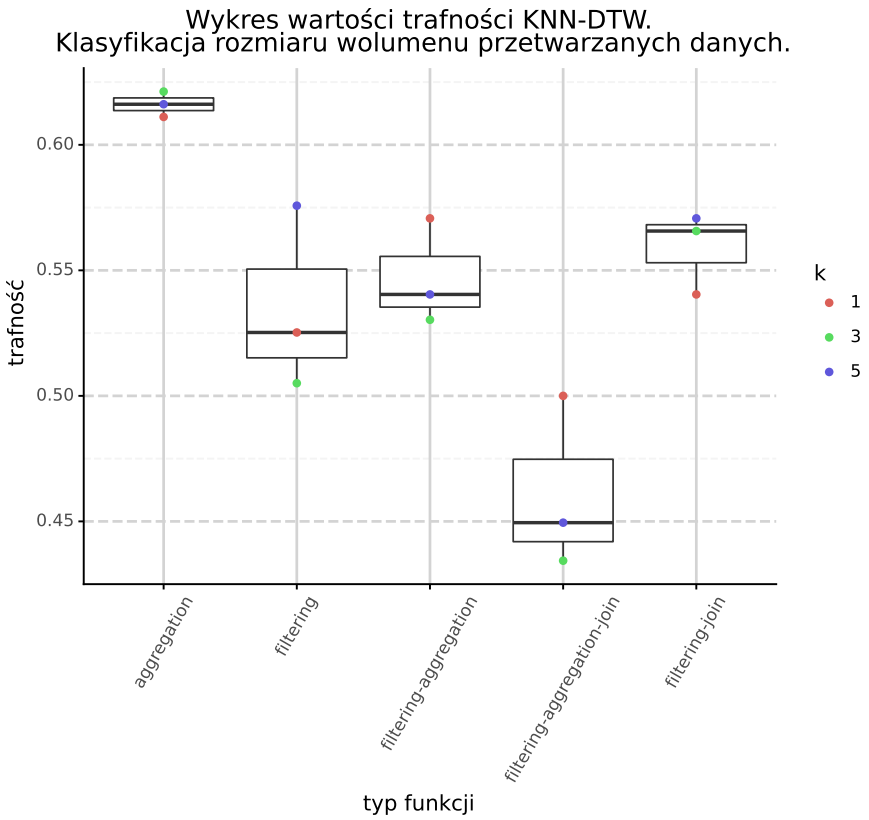
Wyniki metody KNN-DTW

KNN-DTW jako jedyny algorytm korzystał bezpośrednio z podobieństwa szeregów. Uzyskał on najniższe wyniki trafności i F-miary z badanych metod. Najlepiej poradził sobie z klasyfikacją w ramach funkcji typu aggregation uzyskując trafność oraz F-miarę równą średnio 0.616. Najgorzej z kolei klasyfikował w ramach operacji typu filtering-aggregation-join, z trafnością równą średnio 0.461 i F-miarą 0.459. Pozostałe wyniki są wypisane w tablicy 5.21.

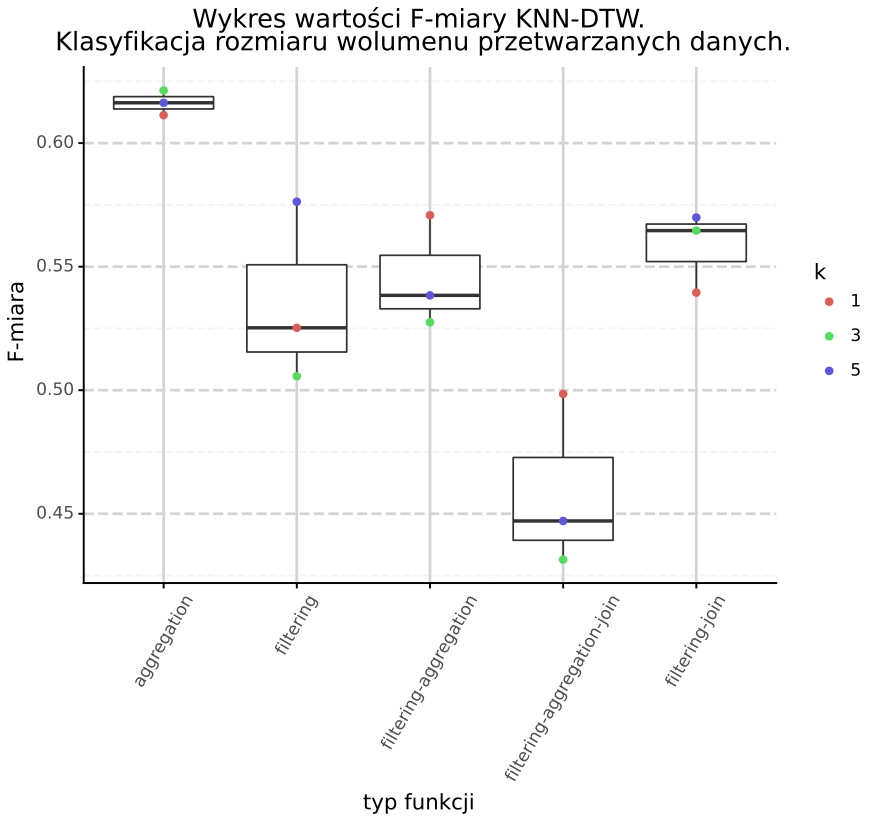
Tablica 5.21: Wyniki eksperymentów algorytmu KNN-DTW dla klasyfikacji wolumenów danych w ramach funkcji

typ funkcji	trafność	f-miara	czas przetwarzania (s)
aggregation	0.616	0.616	3081.419
filtering	0.535	0.536	21.237
filtering-aggregation	0.547	0.546	2197.937
filtering-aggregation-join	0.461	0.459	1905.363
filtering-join	0.559	0.558	79.084

Tym razem wszystkie trafności zbliżone są do wartości 0.5. Najgorszy wynik spadł poniżej 50% co oznacza, że był średnio gorszy od zwykłego losowania. Jako że ten algorytm korzysta z miary podobieństwa DTW, oznaczałoby to, że niezależnie od przetwarzanego wolumenu danych szeregi są podobne w ramach wykonywania tej samej operacji. Czas przetwarzania średnio był relatywnie wysoki, porównując go z poprzednimi algorytmami. Po raz kolejny dla krótszych szeregów ten czas mocno zmalał. Na wykresie pudełkowym 5.42 można zauważyć że wpływ parametru k był niewielki na jakość przetwarzania. Nie ma też jednej wartości, która działała by najlepiej dla każdego typu funkcji.



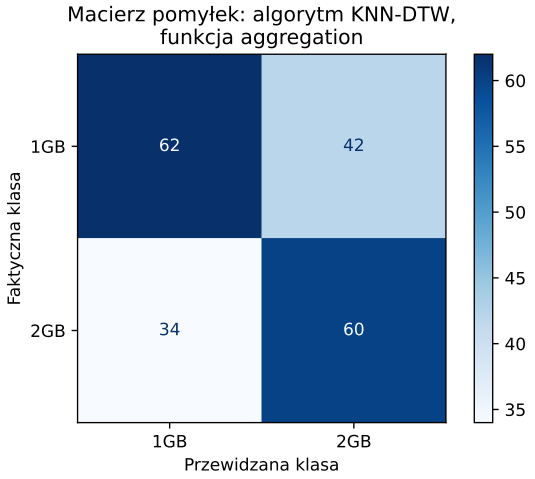
(A) Trafność



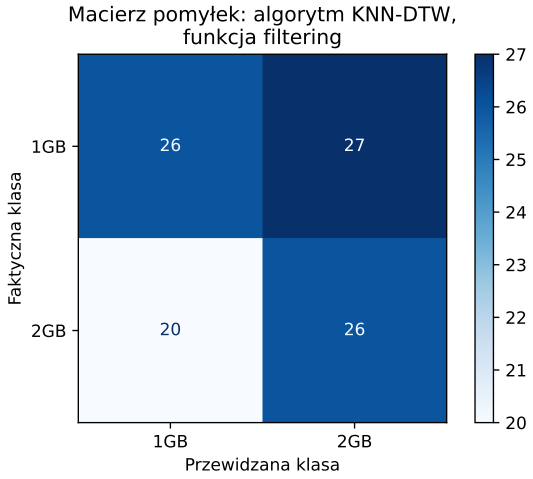
(B) F-miara

Rysunek 5.42: Wykresy pudełkowe trafności oraz F-miary dla algorytmu KNN-DTW.

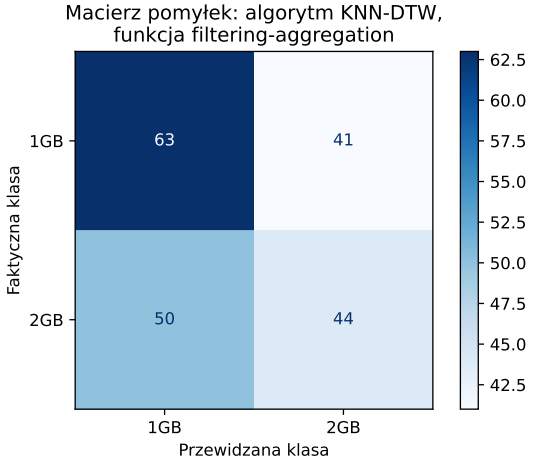
Macierz pomyłek 5.43 po raz kolejny przedstawia, że częściej klasyfikacja popełnia błąd klasyfikacji dla klasy 1GB. Wyjątkiem jest typ funkcji *filtering-aggregation*, podobnie jak w przypadku metody HIVECOTEV2. Tym samym potwierdza to tezę, że szeregi przetwarzające dane o rozmiarze 1GB, są podobne do szeregów przetwarzających dane wielkości 2GB.



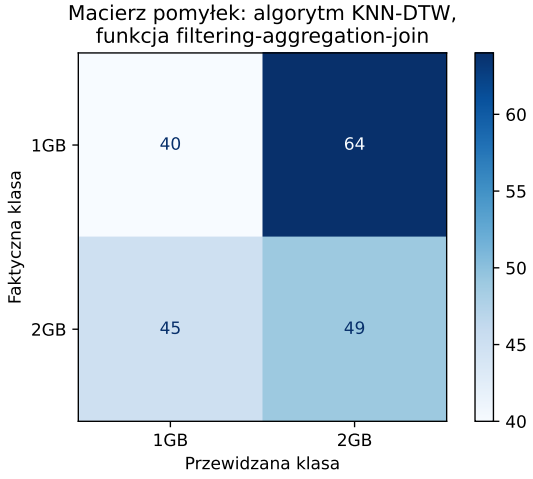
(A) Aggregation



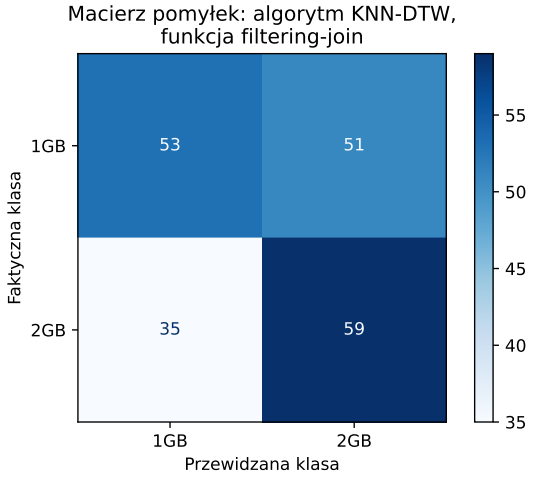
(B) Filtering



(C) Filtering-aggregation



(D) Filtering-aggregation-join

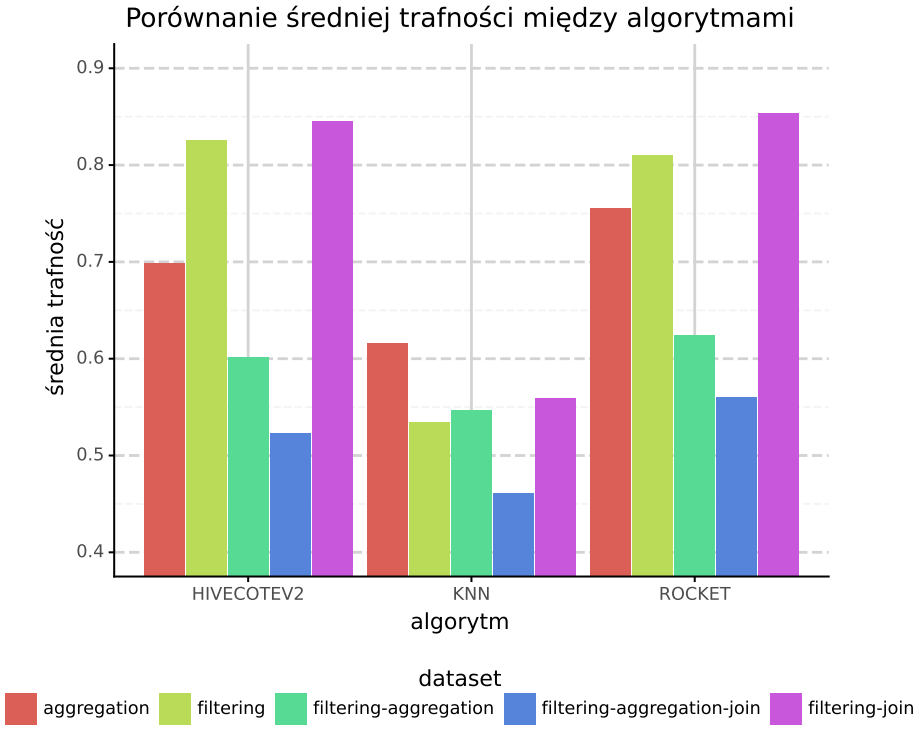


(E) Filtering-join

Rysunek 5.43: Macierze pomyłek dla każdego typu funkcji algorytmu KNN-DTW.

Porównanie i podsumowanie

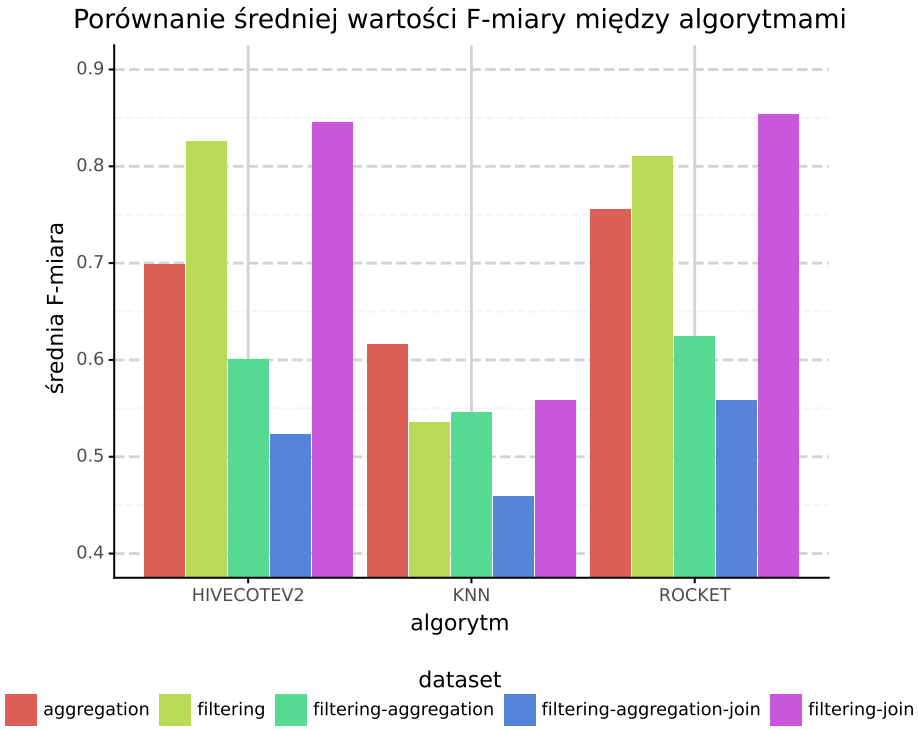
Powyższe eksperymenty miały na celu sprawdzenie czy istnieje podobieństwo między szeregami czasowymi zużycia zasobów dla tych samych funkcji, niezależnie od wielkości przetwarzanych danych. W tym celu wykonano klasyfikacje, gdzie wyniki około 50% trafności wskazywałyby na prawdziwość badanej tezy. Specjalistyczne algorytmy ROCKET oraz HIVECOTEV2 uzyskały wyniki średnio większe od 60% trafności, osiągając nawet wartości tej miary powyżej 80%. Korzystają one jednak z metod, które nie badają stricte podobieństwa, ale raczej inne cechy opisane w rozdziale 2. Należy zwrócić uwagę, że w przypadku klasyfikacji, gdzie klasami były typy funkcji, algorytmy te uzyskały średnio trafność bliską 95%. Pogorszenie wyników o ok. 30% pokazuje więc większą trudność w klasyfikacji wolumenów danych. KNN używając metryki DTW do badania podobieństwa, uzyskał z kolei trafności zbliżone do 50%. Ten algorytm wykorzystuje podobieństwo, co sprawia, że jego wyniki są najbardziej interesujące. Podobnie jak specjalistyczne algorytmy, uzyskał trafność średnio równą 95% przy klasyfikacji typów operacji. Powoduje to, że rezultaty około 50% pokazują, że rozróżnienie czy funkcja przetwarzała dane o rozmiarze 1GB czy 2GB jest problematyczne. Poprzez pokazanie trudności klasyfikacji wolumenów danych wykazano, że szeregi przedstawiające zużycie zasobów podczas przetwarzania danych są do siebie podobne w ramach wykonywania tych samych operacji, niezależnie od wolumenu przetwarzanych danych. Na wykresach od 5.44 do 5.46 można znaleźć zbiorcze wyniki dla wszystkich algorytmów.



Rysunek 5.44: Porównanie średniej trafności klasyfikacji wolumenów danych, z dziesięciu eksperymentów ROCKET oraz HIVECOTEV2 i trzech eksperymentów KNN dla każdego typu funkcji. Wykorzystano szeregi RAM oraz CPU

Tablica 5.22: Zestawienie średniej trafności widocznej na wykresie 5.44

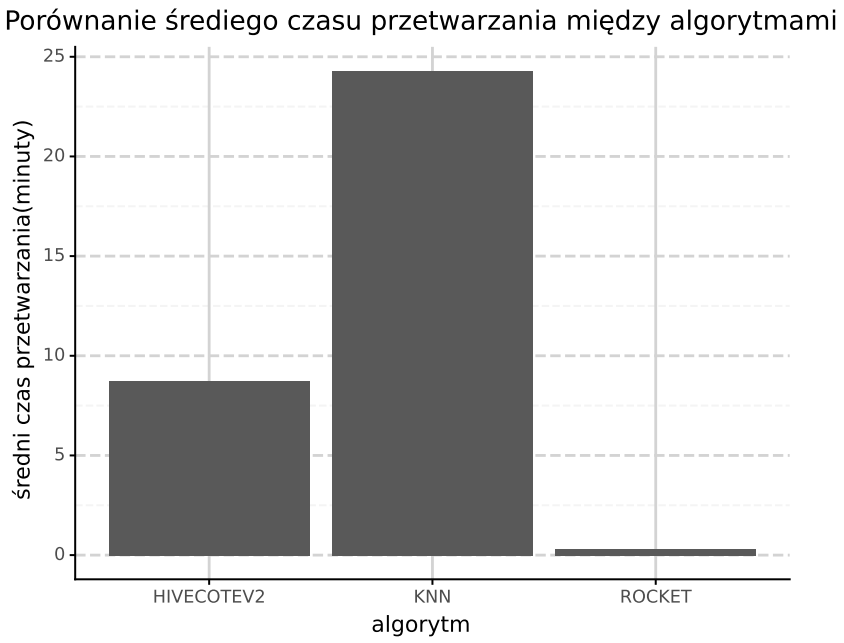
typ funkcji (dataset)	HIVECOTEV2	KNN	ROCKET
aggregation	0.699	0.616	0.756
filtering	0.826	0.535	0.810
filtering-aggregation	0.602	0.547	0.624
filtering-aggregation-join	0.523	0.461	0.560
filtering-join	0.845	0.559	0.854



Rysunek 5.45: Porównanie średniej F-miary klasyfikacji wolumenów danych, z dziesięciu eksperymentów ROCKET oraz HIVECOTEV2 i trzech eksperymentów KNN dla każdego typu funkcji.

Tablica 5.23: Zestawienie trafności widocznej na wykresie 5.45

typ funkcji (dataset)	HIVECOTEV2	KNN	ROCKET
aggregation	0.699	0.616	0.756
filtering	0.826	0.536	0.810
filtering-aggregation	0.601	0.546	0.624
filtering-aggregation-join	0.523	0.459	0.558
filtering-join	0.845	0.558	0.854



Rysunek 5.46: Porównanie średniego czasu przetwarzania klasyfikacji wolumenów danych (1GB oraz 2GB), z wszystkich eksperymentów ROCKET, HIVECOTEV2 i KNN (niezależnie od typu operacji). Parametry komputera wykonującego pomiary: CPU-AMD Ryzen 5 2600x (3.6 GHz, 6 rdzeni, 12 wątków), RAM-2x8GB (3200MHz)

Tablica 5.24: Zestawienie średnich czasów wykonywania widocznych na wykresie 5.46

algorytm	średni czas przetwarzania (minuty)	odchylenie standardowe (minuty)
HIVECOTEV2	14.09	2.47
KNN	265.46	22.59
ROCKET	1.07	0.16

Rozdział 6

Podsumowanie eksperymentów

W tym rozdziale podsumowano eksperymenty korzystające z algorytmów obliczających podobieństwo szeregów czasowych oraz eksperymenty klasyfikujące szeregi czasowe przy pomocy uczenia maszynowego. Następnie zostały porównane te dwa podejścia poprzez porównanie algorytmu DTW z klasyfikatorem KNN-DTW.

6.1 Podobieństwo szeregów

Eksperymenty do których nawiązuje poniższa sekcja zostały opisane w działach 5.1.1 i 5.2.1.

Tablica 6.1 przedstawia podsumowanie obserwacji dla algorytmów liczenia odległości między szeregami dla danych zużycia CPU. Na jej podstawie można uznać algorytm Dynamic Time Warping za najlepszy z trzech przetestowanych. Wykorzystanie odległości euklidesowej również nie byłoby całkowicie złym pomysłem. Analizując zestawienie średnich odległości euklidesowych dla CPU (rysunek 5.3) można podejrzewać, że klasyfikator zbudowany na podstawie tej miary miałby problem z odróżnieniem agregacji i agregacji-filtracji, gdyż zachowują się one podobnie w stosunku do każdego z typów funkcji. Algorytm Longest Common Subsequence poradził sobie najgorzej.

Tablica 6.1: Zestawienie obserwacji na danych CPU dla algorytmów podobieństwa.

Opis	Odległość Euklidesowa	Dynamic Time Warping	Longest Common Subsequence
typy funkcji, dla których najmniejszą średnią odległość mają te same typy - dane nieprzetworzone	filtracja, filtracja-agregacja, filtracja-agregacja-join	filtracja, filtracja-agregacja, filtracja-agregacja-join, filtracja-join	filtracja, filtracja-agregacja
typy funkcji, dla których najmniejszą średnią odległość mają te same typy - dane wygładzone	agregacja, filtracja, filtracja-agregacja, filtracja-agregacja-join	agregacja, filtracja, filtracja-agregacja, filtracja-agregacja-join, filtracja-join	filtracja, filtracja-agregacja
najmniejsza odległość jest między filtracjami	tak	tak	tak
cztery najmniejsze średnie odległości zaobserwowano dla porównań filtracja:filtracja, filtracja-join:filtracja-join i filtracja:filtracja-join	nie	nie	tak

Tablica 6.2 pokazuje podsumowanie obserwacji dla algorytmów liczenia odległości między szeregami dla danych zużycia RAM. Jedynymi typami funkcji, dla których najmniejszą średnią odległość miały te same typy są filtracja-agregacja-join i filtracja-join. Bez problemu można za to odróżnić filtrację i filtrację-join od pozostałych typów. Dodatkowo w odległości euklidesowej i LCSS występuje zależność, że najbliższym typem dla filtracji-agregacji jest filtracja-agregacja-join.

Tablica 6.2: Zestawienie obserwacji dla algorytmów RAM

Opis	Odległość Euklidesowa	Dynamic Time Warping	Longest Common Subsequence
typy funkcji, dla których najmniejszą średnią odległość mają te same typy - dane nieprzetworzone	filtracja-agregacja-join, filtracja-join	filtracja-agregacja-join, filtracja-join	filtracja-agregacja-join, filtracja-join
typy funkcji, dla których najmniejszą średnią odległość mają te same typy - dane wygładzone	filtracja-agregacja-join, filtracja-join	filtracja-agregacja-join, filtracja-join	filtracja-agregacja-join, filtracja-join
cztery najmniejsze średnie odległości zaobserwowano dla porównań filtracja:filtracja, filtracja-join:filtracja-join i filtracja:filtracja-join	tak	tak	tak
najbliższym typem dla filtracji-agregacji jest filtracja-agregacja-join	tak	nie	tak
największe średnie odległości zaobserwowano między filtracją i filtracją-join a pozostałymi typami	tak	nie	nie

Przeanalizowano również wpływ rozmiaru danych na odległości między szeregami czasowymi. Zaobserwowano, że dla algorytmu DTW i odległości euklidesowej, rozmiar danych nie wpływa znacząco na podobieństwo zużycia CPU. Ma jednak wyraźny wpływ na podobieństwo zużycia RAM. Algorytm LCSS poradził sobie najgorzej, jednak odróżnienie filtracji i filtracji-join od reszty nie powinno być problemem niezależnie od rozmiaru.

6.2 Uczenie maszynowe

Drugą metodą wykorzystaną w pracy było uczenie maszynowe, które zostanie opisane w niniejszej sekcji. Klasyfikacja ma posłużyć jako dowód na to, że istnieje pewne podobieństwo szeregów czasowych w ramach wykonywania tej samej operacji. Wysoka trafność i F-miara uzyskana przy pomocy danych wykorzystanych w pracy, potwierdzą tą tezę. Należy przypomnieć, że w zbiorach wykorzystywanych w eksperymentach znajdowały się przebiegi wykonywane na różnych rozmiarach danych (w tym przypadku 1GB lub 2GB). Wszystkie zbadane algorytmy klasyfikujące osiągnęły trafność oraz F-miarę powyżej 95%. Dokładnie wyniki i macierze pomyłek znajdują się w rozdziale 5.

Szczególną uwagę należy zwrócić na metodę *ROCKET*, gdyż osiągnęła najlepszą trafność równą średnio 0.998. Algorytm *HIVECOTEV2* też miał potencjał na tak wysoki wynik, jednak został on ograniczony czasowo. Bez limitu algorytm ten wykonywał się długo (nawet po trzech godzinach przetwarzania nie zwracał wyników), co spowodowało ograniczenie go parametrem wejściowym do pięciu minut. Dzięki temu zwracał rezultaty szybciej, po średnio ok. piętnastu minutach. Jego

ostateczny wynik zatem mógł być lepszy, jednak średnia trafność 0.962 mimo tego ograniczenia jest bardzo wysoka. Wspomniane algorytmy poradziły sobie również z problemem rozróżnienia wolumenów przetwarzanych danych. ROCKET jak i HIVECOTEV2 średnio uzyskały trafność równą średnio ok. 70%. Trafność ta jest niższa od trafności eksperymentów badających typy (średnio niższa średnio o ok. dwadzieścia pięć punktów procentowych), jednak wciąż można stwierdzić, że rozmiary przetwarzanych danych były dla nich rozróżnialne. Algorytmy te wykorzystywały inne mechanizmy niż bezpośrednie podobieństwo do klasyfikacji szeregów. Powoduje to, że ich bezpośrednie porównanie z metodami z sekcji 6.1 może być mało interesujące.

Ostatnim użytym algorytmem klasyfikacyjnym był *KNN*, wykorzystujący Dynamic Time Warping jako miarę podobieństwa między szeregami. Ta klasyczna metoda wykazała najwyższą trafność oraz F-miarę równą średnio 0.964. Wynik ten jest porównywalny z opisanymi powyżej specjalistycznymi algorytmami. Biorąc to pod uwagę, jego wyniki dla rozróżnienia wolumenów przetwarzanych danych są raczej zbliżone 55%. Oznaczałoby to, że wykorzystując samo podobieństwo trudno rozróżnić przetwarzanie 1GB od 2GB, zachowując przy tym poprawne wyniki przy rozróżnianiu różnych typów operacji. KNN-DTW, jako jedyny zbadany algorytm wykorzystujący podobieństwo, został porównany w sekcji 6.3 z wynikami algorytmu Dynamic Time Warping.

6.3 Porównanie podejść

6.3.1 Porównanie eksperymentów badających podobieństwo typu funkcji

W tej sekcji porównano wyniki uzyskane za pomocą algorytmu Dynamic Time Warping i za pomocą algorytmu KNN. W KNN wykorzystany był Dynamic Time Warping, więc ich porównanie pozwala na walidację wyników.

Za sukces można uznać wynik algorytmu DTW. Pokazuje on, że podobne typy mają średnio mniejsze odległości między sobą, niż te same typy porównane z innymi. Można to zaobserwować na rysunku 5.14. Wykorzystanie KNN również dało oczekiwane rezultaty. Jego trafność to aż 96.4%.

Tablica 6.3 obrazuje porównanie czułości algorytmu KNN-DTW z różnicą procentową między średnią odległością dla danego typu, a średnią odległością porównania tego typu z pozostałymi. Różnicę procentową opisano wzorem:

$$\frac{avgSameType - avgDifferentType}{avgSameType} * 100\%,$$

gdzie `avgSameType` to średnia odległością dla danego typu, a `avgDifferentType` to średnia odległość porównania tego typu z pozostałymi.

Tablica 6.3: Różnica między danym typem, a resztą

Typ funkcji	Czułość algorytmu KNN-DTW	Różnica między danym typem, a resztą
aggregation	84.3%	-33.87%
filtering	99.0%	-156.33%
filtering-aggregation	97.7%	-63.34%
filtering-aggregation-join	100%	-51.17%
filtering-join	93.9%	-65.51%

Z tablicy można odczytać, że KNN najgorzej radzi sobie z rozpoznawaniem agregacji. Również różnica między odległością dla porównywania agregacji z agregacją, a porównywania agregacji z

innymi typami jest najmniejsza. Dla pozostałych typów funkcji nie rozpoznano podobnej korelacji, jednak wszystkie z pozostałych typów dają znacznie lepsze wyniki - powyżej 93%.

6.3.2 Porównanie eksperymentów badających podobieństwo rozmiarów przetwarzanych danych

Porównując wyniki eksperymentów dotyczących typów funkcji pokazano, że algorytm Dynamic Time Warping uzyskuje poprawne wyniki przetwarzając badany zbiór danych. W przypadku gdy przetwarzanie 1GB jest podobne do przetwarzania 2GB, algorytm DTW powinien wykazać to podobieństwo. Eksperymenty wykorzystujące DTW wykazały niewielkie różnice odległości między tymi samymi, a różnymi wolumenami danych, w przypadku szeregów przestawiających zużycie CPU. Średnio wynosiła ona około 4.8% dla algorytmu DTW. Korzystając z uczenia maszynowego, klasyfikacja nie otrzymywała wysokiej trafności. Średnio osiągała ona 55% korzystając z klasyfikatora KNN-DTW. Jest to sygnał oznaczający trudność w rozróżnieniu szeregów przetwarzających 1GB od 2GB, co mogłoby oznaczać że są one do siebie zbyt podobne. W tablicy 6.4 znajduję się porównanie trafności algorytmu KNN, z procentową różnicą między podobieństwem tych samych, a różnych rozmiarów danych(opisane w sekcji 5.2.1).

Tablica 6.4: Porównanie trafności i procentowej różnicy podobieństw wolumenów danych w ramach tego samego typu operacji

Typ funkcji	Trafność KNN-DTW	Podobieństwo DTW
aggregation	0.616	11.56%
filtering	0.535	3.86%
filtering-aggregation	0.547	4.18%
filtering-aggregation-join	0.461	0.38%
filtering-join	0.559	2.37%

W tablicy 6.4 można zauważyć, że im większy procent różnych podobieństw, tym wyższa trafność. Wyniki klasyfikacji są średnio zbliżone do 50% trafności, tak samo różnica podobieństw jest stosunkowo niewielka; bliska średnio 5%. Pokazuje to, że oba podejścia dały spójne wyniki wskazujące na podobieństwo szeregów czasowych w ramach tych samych typów operacji, niezależnie od wielkości danych.

Rozdział 7

Wnioski i perspektywy badawcze

Tworząc aplikacje przetwarzające dane, programiści mogą zdefiniować UDF (User Defined Functions), których zawartość często może być określona jako black box. Takiego typu funkcje trudno zoptymalizować, z względu na brak wiedzy o operacjach w nich zawartych. Niniejsza praca podejmuje próbę znalezienia najefektywniejszych metod, które umożliwiłby **określenie typów funkcji na podstawie szeregów czasowych reprezentujących zużycie zasobów komputerowych przez procesy przetwarzania danych**. W pracy zostały zbadane szeregi czasowe przedstawiające zużycie procesora i pamięci operacyjnej podczas działania aplikacji przetwarzających dane. Te programy wykonywały operacje takie jak agregacja, filtracja, połączenia i ich różne kombinacje, na danych, które miały rozmiar 1GB lub 2GB. Celem było sprawdzenie czy te typy funkcji są rozróżnialne oceniając podobieństwo tych szeregów czasowych. Wykonane zostało to przy pomocy algorytmów obliczających podobieństwo między szeregami czasowymi oraz korzystając z algorytmów uczenia maszynowego.

Wyniki eksperymentów przestawione w rozdziale 5. i podsumowane w rozdziale 6. wykazują, że występuje podobieństwo w ramach typów funkcji w badanym zbiorze danych. Dodatkowo analiza porównawcza w sekcji 6.3 pokazała, że wyniki są spójne dla obu podejść, czyli algorytmów obliczających podobieństwo oraz klasyfikacji przy pomocy uczenia maszynowego. Można to zauważyć szczególnie poprzez porównanie wyników eksperymentów, gdzie obliczając podobieństwo wykorzystano algorytm **Dynamic Time Warping**, a klasyfikacja wykonana była przy pomocy KNN z wykorzystaniem tego samego algorytmu (DTW) jako miara podobieństwa. Na podstawie tego można wywnioskować, że **w przypadku badanego zbioru danych, jest możliwe rozróżnienie typów funkcji.**

Decydując się na najlepszą metodę do rozróżniania typów funkcji, wybrano podejście z wykorzystaniem uczenia maszynowego. Samo podobieństwo, nie było do końca deterministyczne. Najlepiej poradził sobie algorytm ROCKET, który nie korzysta jedynie podobieństwa, lecz z cech wydobytych przy pomocy losowych kerneli konwolucyjnych. Dodatkowo uczenie maszynowe w prosty sposób wykorzystuje przebiegi zużycia RAM w połączeniu z szeregami zużycia CPU. W przypadku badania podobieństwa problematyczne mogłoby być spójne użycie szeregów czasowych przedstawiających zużycie dwóch różnych zasobów. **Tym samym proponowaną metodą do rozróżniania typów UDF przetwarzających dane jest uczenie maszynowe, a dokładniej algorytm ROCKET.**

Omawiając CPU i zastosowanie na nim algorytmów podobieństwa, różnice między odległościami dla danych o rozmiarze 1GB, a 2GB (tablica 5.11) są znacznie mniejsze od różnicy między średnią odległością dla danego typu, a średnią odległością porównania tego typu z pozostałymi (tablica 6.3). **Można więc założyć, że nie powinny one zbytnio wpływać na rozróżnie-**

nie typów. Brak dużej różnicy między rozmiarami danych udowodniono dla algorytmu KNN. Próbował on klasyfikować, jaki rozmiar danych był używany przy powstaniu danego szeregu czasowego, co się nie udało (wyniki były niemal losowe). Trochę inaczej ma się kwestia zużycia pamięci RAM. Jako, że pamięć RAM była wykorzystywana przede wszystkim do załadowania i przechowania danych, to wielkość tych danych musiała mieć wpływ na jej zużycie. Jak pokazuje tablica 5.13 różnice były bardzo duże. Algorytmy uczenia maszynowego, które nie korzystają z algorytmów podobieństwa (ROCKET, HIVECOTE2) radziły sobie lepiej ze znalezieniem różnicy między szeregami czasowymi o różnych rozmiarach danych.

Przechodząc do wyników dla konkretnych typów funkcji, można zauważyć, że agregacja to specyficzna operacja. Podczas badania podobieństwa, agregacja wykazywała bardzo małe różnice w wynikach w porównaniu z innymi typami. Ta zależność jest wyraźnie zarysowana w tablicy 6.3. Wykres 5.15 pokazuje, że używając najlepszego z badanych algorytmów, trudno jest rozróżnić agregacje od reszty typów. Każda inna operacja ją przysłania. Analogicznie uczenie maszynowe popełniało najwięcej błędów, gdy próbowało klasyfikować agregacje. Jest to widoczne w każdym z badanych algorytmów. W przypadku metody ROCKET, agregacja została przypisana filtracjo-agregacji (dla zestawu danych Default był to jedyny błąd). Najlepszy wynik HIVECOTEV2 popełnił aż 15 z 17 błędów związanych z klasyfikacją agregacji. Podobnie w przypadku KNN-DTW, 13 z 13 błędów było związanych z agregacją. **Można z tego wywnioskować, że zdecydowanie najtrudniej jest wykryć przetwarzanie wykorzystujące agregacje w funkcjach UDF.**

Dalej przyglądając się wynikom, można zauważyć, że filtracja jest bardzo podobna do innej filtracji (takiej która nie zawiera agregacji), jeżeli spojrzymy na szeregi przedstawiające RAM (przykładowo rysunki od 5.19 do 5.22). **Wskazuje to na to, że RAM jest odpowiednim wskaźnikiem tego, czy dana operacja wykonuje filtrowanie bez agregowania danych.** Niestety w przypadku algorytmów uczenia maszynowego nie mamy informacji w jakim stopniu zostały wykorzystane szeregi przedstawiające zużycie pamięci RAM, jednak można się domyślać, że pomogły z klasyfikacją tych właśnie typów. Ta zależność nie jest wzajemna, gdyż podobieństwo między szeregami zawierającymi agregacje nie jest tak wysokie. Z tego powodu jednym słusznym zastosowaniem zużycia pamięci operacyjnej, w tym zbiorze danych, **jest identyfikacja filtrowania bez agregowania.** Aby rozróżnić filtracje od filtracji-join trzeba wykorzystać już szeregi przedstawiające zużycie CPU.

Wszystkie algorytmy podobieństwa charakteryzują się tym, że jeżeli podano szereg czasowy nie znając wykonywanej operacje, to samo wybranie typu, do którego odległość jest najmniejsza nie wystarczyłoby do prawidłowego odgadnięcia tej operacji. Jak można zauważyć na rysunku 5.16, obliczony dystans dla danego typu, pokrywa się częściowo z pozostałymi. Odległości wykazywały się również innymi charakterystykami, np. dla DTW dla danych nieprzetworzonych zużycia CPU, średnia odległość między filtracją-agregacją-join, a jakimkolwiek typem jest bardzo duża, w porównaniu do pozostałych pomiarów. Dlatego można podejrzewać, że **mając wiedzę o odległości danego szeregu względem każdego z typów funkcji, można zaklasyfikować go do któregoś z nich na podstawie średnich odległości między znanymi typami.**

Powodem dla którego algorytm Longest Common Subsequence działał mniej efektywnie od pozostałych, jeśli chodzi o dane zużycia CPU, może być to, że uznawał on wartości zero, które były znaczną częścią każdego szeregu za wspólny podciąg. Dlatego odległości były bardzo małe.

Perspektywy badawcze

Powyższe wnioski pokazują, że jest możliwe odróżnienie typów wykonywanych operacji przetwarzających dane w ramach UDF'ów. Na ich podstawie można szukać dalszych zastosowań w

świecie nauki lub rozszerzyć je w celu bardziej deterministycznych wyników.

Jeden ze sposobów dalszych badań, może polegać na zebraniu innego zbioru danych, z większą różnicą rozmiarów przetwarzanych danych. Tego typu eksperymenty utwierdziłyby przekonanie, że rozmiar nie ma większego wpływu na identyfikacje typów operacji. 1GB oraz 2GB wydają się być zbyt małą różnicą.

Kolejną rzeczą, nad którą możnaby się zastanowić to wybranie innych cech charakteryzujących wykorzystanie funkcji UDF. Podczas gdy zużycie CPU było bardzo dobrym wyborem, tak zużycie RAM niekoniecznie. W celu uzyskania większej dokładności rozróżniania typów, możnaby użyć dane dotyczące np. ruchu sieciowego w przypadku użycia klastra lub liczby operacji wejścia/wyjścia.

Należałoby też wykorzystać fakt, że jest możliwe odróżnienie zbadanych operacji. UDF'y w normalnych warunkach nie mogą być optymalizowane ze względu na brak wiedzy o typie przetwarzania w ich środku. Jeżeli udałoby się przewidzieć typy wykonywanych operacji wewnątrz UDF'ów, należałoby sprawdzić, czy jest możliwość przyspieszenia ich wykonywania.

Wyniki uzyskane w niniejszej pracy zdecydowanie wskazują na to, że może to być ciekawy kierunek dalszych badań. Mogłyby one posłużyć w udoskonaleniu systemów przetwarzających duże wolumeny danych przy pomocy User Defined Function.

Literatura

[1] Wei Tang, Narayan Desai, Daniel Buettner, and Zhiling Lan. Job scheduling with adjusted runtime estimates on production supercomputers. *Journal of Parallel and Distributed Computing*, 73(7):926–938, 2013.

[2] Sarunya Pumma, Wu chun Feng, Phond Phunchongharn, Sylvain Chapeland, and Tiranee Achalakul. A runtime estimation framework for alice. *Future Generation Computer Systems*, 72:65–77, 2017.

[3] Alvaro Brandon, María Pérez, Smrati Gupta, and Victor Muntés-Mulero. Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems*, 86, 2017.

[4] Hubert Krzyżanowski and Łukasz Pięta. Odkrywanie charakterystyk user defined functions. Praca Magisterska, Politechnika Poznańska, 2021.

[5] Michał Bodziony, Hubert Krzyżanowski, Łukasz Pięta, and Robert Wrembel. On discovering semantics of user-defined functions in data processing workflows. In *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, In conjunction with the ACM SIGMOD/PODS Conference, BiDEDE@SIGMOD*, pages 7:1–7:6. ACM, 2021.

[6] Donald J Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *KDD workshop*, volume 10, pages 359–370. The Association for the Advancement of Artificial Intelligence and The MIT Press, 1994.

[7] Gautam Das, Dimitrios Gunopulos, and Heikki Mannila. Finding similar time series. In *Principles of Data Mining and Knowledge Discovery*, pages 88–100. Springer, 1997.

[8] C.-S. Perng, H. Wang, S.R. Zhang, and D.S. Parker. Landmarks: a new model for similarity-based pattern querying in time series databases. In *Proceedings of the International Conference on Data Engineering*, pages 33–42, 2000.

[9] Pratap Chandra Sen, Mahimarnab Hajra, and Mitadru Ghosh. Supervised classification algorithms in machine learning: A survey and review. In *Emerging Technology in Modelling and Graphics*, pages 99–111. Springer, 2020.

[10] Morzy Tadeusz. *Eksploracja danych. Metody i algorytmy.* Wydawnictwo Naukowe PWN S.A., 2013.

[11] Angus Dempster, François Petitjean, and Geoffrey I. Webb. ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels. *Data Mining and Knowledge Discovery*, 34(5):1454–1495, 2020.

[12] Matthew Middlehurst, James Large, Michael Flynn, Jason Lines, Aaron Bostrom, and Anthony Bagnall. Hive-cote 2.0: a new meta ensemble for time series classification. *Machine Learning*, 110(11):3211–3243, 2021.

[13] Vivek Mahato, Martin O'Reilly, and Pádraig Cunningham. A comparison of k-nn methods for time series classification and regression. In *ASUS Intelligent Cloud Services*, 2018.

[14] M. Shanker, M.Y. Hu, and M.S. Hung. Effect of data standardization on neural network training. *Omega*, 24(4):385–397, 1996.

[15] Bartosz Ciesielski and Anna Lehnardt. Notatniki oraz dane wykorzystywane w eksperymentach. `github.com/Put-MasterThesis-TimeSeries-Sim/black-box-udf-timeseries-classification`, 2022.

[16] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[17] T Giorgino. Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw package. *JSS Journal of Statistical Software*, 31, 2009.

[18] Wannes Meert, Toon Van Craenendonck Hendrickx, Kilian, and Robberechts Pieter. Dtaidistance (version v2). zenodo. `http://doi.org/10.5281/zenodo.5901139`.

[19] Romain Tavenard, Johann Faouzi, Gilles Vandewiele, Felix Divo, Guillaume Androz, Chester Holtz, Marie Payne, Roman Yurchak, Marc Rußwurm, Kushal Kolar, and Eli Woods. Tslearn, a machine learning toolkit for time series data. *Journal of Machine Learning Research*, 21(118):1–6, 2020.

[20] Markus Löning, Franz Király, Tony Bagnall, Matthew Middlehurst, Sajaysurya Ganesh, Oastler. George, Jason Lines, ViktorKaz, Martin Walker, Lukasz Mentel, Stanislav Khrapov, RNKuhns, chrisholder, Leonidas Tsaprounis, Taiwo Owoseni, Patrick Rockenschaub, jesellier, danbartl, eenticott shell, Guzal Bulatova, Ciaran Gilbert, Lovkush, Kejsi Take, Svea Marie Meyer, AidenRushbrooke, Mirae Parker, Patrick Schäfer, oleskiewicz, Yi-Xuan Xu, and Afzal Ansari. alan-turing-institute/sktime: v0.11.4. `https://github.com/alan-turing-institute/sktime`, 2022.

[21] Markus Löning, Anthony J. Bagnall, Sajaysurya Ganesh, Viktor Kazakov, Jason Lines, and Franz J. Király. sktime: A unified interface for machine learning with time series. *CoRR*, abs/1909.07872, 2019.

[22] Josif Grabocka, Nicolas Schilling, Martin Wistuba, and Lars Schmidt-Thieme. Learning time-series shapelets. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 392–401. Association for Computing Machinery, 2014.

[23] The pandas development team. pandas-dev/pandas: Pandas. `https://doi.org/10.5281/zenodo.3509134`, 2020.

Dodatek A

Repozytorium

Repozytorium na stronie `https://github.com/Put-MasterThesis-TimeSeries-Sim/black-box-udf-timeseries-classification` zawiera oprogramowanie stworzone przez zespół. W folderze *Notebooks* można znaleźć przetwarzanie danych oraz przeprowadzone eksperymenty zawarte w plikach *Jupyter Notebook*. W folderze *Raw* znajdują się surowe dane potrzebne do uruchomienia wszystkich kolejnych notatników. Katalogi *Prepared* oraz *Organized* zawierają przetworzone dane. Folder *Plots* zawiera wszelkie grafy wykorzystane w pracy zbudowane przy pomocy notatników. Plik *readme.md* zawiera instrukcje uruchomienia przygotowanego kodu. Katalog *Thesis* zawiera pliki źródłowe plików .tex napisanej pracy dyplomowej.

© 2022 Anna Lehnhardt, Bartosz Ciesielski

Instytut Informatyki, Wydział Informatyki i Telekomunikacji
Politechnika Poznańska

Skład przy użyciu systemu LaTeX na platformie Overleaf.